INTRODUCTION TO PROCESS

A process is an instance of a program running in a computer. It is close in meaning to task, a term used in some operating systems. In UNIX and some other operating systems, a process is started when a program is initiated (either by a user entering a shell command or by another program). Like a task, a process is a running program with which a particular set of data is associated so that the process can be kept track of. An application that is being shared by multiple users will generally have one process at some stage of execution for each user. A process is basically a program in execution. In computing, a process is the instance of a computer program that is being executed. It contains the program code and its activity. Further A process is defined as an entity which represents the basic unit of work to be implemented in the system. A process can initiate a sub-process, which is a called a child process (and the initiating process is sometimes referred to as its parent). A child process is a replica of the parent process and shares some of its resources, but cannot exist if the parent is terminated. Processes can exchange information or synchronize their operation through several methods of inter process communication (IPC will be covered later).

To put it in simple terms, we write our computer programs in a text file and when we execute this program, it becomes a process which performs all the tasks mentioned in the program. When a program is loaded into the memory and it becomes a process, it can be divided into four sections stack, heap, text and data. The following image shows a simplified layout of a process inside main memory.

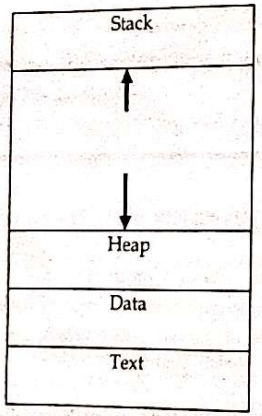


Fig 3.1: layout of a process inside main memory

- **Stack:** The process Stack contains the temporary data such as method/function parameters return address and local variables.
- **Heap:** This is dynamically allocated memory to a process during its run time.
- **Text:** This includes the current activity represented by the value of Program Counter and the contents of the processor's registers.
- **Data:** This section contains the global and static variables.

PROGRAM

Program is an executable file containing the set of instructions written to perform a specific job on your computer. For example, chrome.exe is an executable file containing the set of instructions written so that we can view web pages. **notepad.exe** is an executable file containing the set of instructions which help us to edit and print the text files. Programs are not stored on the primary memory in your computer. They are stored on a disk or a secondary memory on your computer. They are read into the primary memory and executed by the kernel. A program is sometimes referred as passive entity as it resides on a secondary memory.

A program is a piece of code which may be a single line or millions of lines. A computer program is usually written by a computer programmer in a programming language. For example, here is a simple program written in C programming language

```
#include <stdio.h>
int main()
{
        printf("Hello, World! \n");
        return 0;
}
```

A computer program is a collection of instructions that performs a specific task when executed by a computer. When we compare a program with a process, we can conclude that a process is a dynamic instance of a computer program. A part of a computer program that performs a well-defined task is known as an algorithm. A collection of computer programs, libraries and related data are referred to as a software.

Some examples of computer programs:

- Operating system
- A web browser like Mozilla Firefox and Apple Safari can be used to view web pages on the Internet.
- An office suite can be used to write documents or spreadsheets.
- Video games are computer programs.

A computer program is written by a programmer. It is very difficult to write in the ones and zeroes of machine code, which is what the computer can read, so computer programmers write in a programming language, such as BASIC, C, or Java. Once it is written, the programmer uses a compiler to turn it into a language that the computer can understand.

There are also bad programs, called malware, written by people who want to do bad things to a computer. Some are spyware, trying to steal information from the computer. Some try to damage the data stored on the hard drive. Some others send users to web sites that offer to sell them things. Some are computer viruses or ransomware.

Differences between program and process

a. A program is a definite group of ordered operations that are to be performed. On the other hand, an instance of a program being executed is a process.

b. The nature of the program is passive as it does nothing until it gets executed whereas process is dynamic or active in nature as it is an instance of executing program and performs the specific action.

c. A program has a longer lifespan because it is stored in the memory until it is manually deleted while a process has a shorter and limited lifespan because it gets terminated after the completion of the task.

d. The resource requirement is much higher in case of a process; it could need processing memory, I/O resources for the successful execution. In contrast, a program just requires memory for storage.

PROCESS MODEL

In process model, all the runnable software on the computer is organized into a number of sequential processes. Each process has its own virtual Central Processing Unit (CPU).The real Central Processing Unit (CPU) switches back and forth from process to process. This work of switching back and forth is called multi-programming. A process is basically an activity. It has a program, input, output, and a state. The operating system must need a way to make sure that all the essential processes exist. There are the following four principal events that cause the processes to be created.

- System initialization
- Execution of a process creation system call by a running process
- A user request to create a new process
- Initiation of a batch work

Generally, there are some processes that are created whenever an operating system is booted. Some of those are foreground processes and others are background processes.

- Foreground process is the process that interacts with the computer users or computer programmers.
- Background processes have some specific functions.

In UNIX system, the ps program can be used to list all the running processes and in windows, the task manager is used to see what programs are currently running into the system. In addition to the processes that are created at the boot time, new processes can also be created. Sometime a running process will issue the system calls just to create one or more than one new processes to help it to do its work.

User can start a program just by typing the command of the program on the command prompt (CMD) or just by doing the double click on the icon of that program. When a process has been created, it starts running and does its work. The new process will terminate generally due to one of the following conditions, described in the table given below.

Condition	Description
Normal Exit	In Normal exit, process terminates because they have done their work successfully
Error Exit	In error exit, the termination of a process is done because of an error caused by the process, sometimes due to the program bug
Fatal Exit	In fatal exit, process terminates because it discovers a fatal error
Killed by other process	In this reason or condition, a process might also terminate due to that it executes a system call that tells the operating system (OS) just to kill some other process

In some computer systems when a process creates another process, then the parent process and child process continue to be associated in certain ways. The child process can itself creates more processes that form a process hierarchy.

PROCESS LIFE CYCLE

When a process executes, it passes through different states. These stages may differ in different operating systems, and the names of these states are also not standardized. The operating system maintains management information about a process in a process control block (PCB). Modern operating systems allow a process to be divided into multiple threads of execution, which share all process management information except for information directly related to execution. This information is held in a thread control block (TCB). Threads in a process can execute different parts of the program code at the same time. They can also execute the same parts of the code at the same time, but with different execution state:

- They have independent current instructions; that is, they have (or appear to have) independent program counters.
- They are working with different data; that is, they are (or appear to be) working with independent registers.

In general, a process can have one of the following five states at a time.

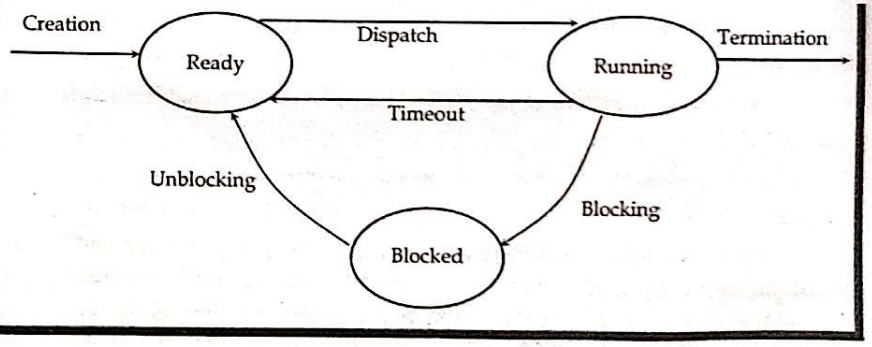


Fig 3.2: Process life cycle

1. **Start:** This is the initial state when a process is first started/created
2. **Ready:** The process is waiting to be assigned to a processor. Ready processes are waiting t have the processor allocated to them by the operating system so that they can run. Proces may come into this state after Start state or while running it by but interrupted by th scheduler to assign CPU to some other process.
3. **Running:** Once the process has been assigned to a processor by the OS scheduler, the proces state is set to running and the processor executes its instructions.
4. **Waiting:** Process moves into the waiting state if it needs to wait for a resource, such as waiting for user input, or waiting for a file to become available.
5. **Terminated or Exit:** Once the process finishes its execution, or it is terminated by th operating system, it is moved to the terminated state where it waits to be removed from main memory.

Suspended Processes

Characteristics of suspend process

- Suspended process is not immediately available for execution.
- The process may or may not be waiting on an event.
- For preventing the execution, process is suspended by OS, parent process, process itself and an agent.
- Process may not be removed from the suspended state until the agent orders th removal.

Note: Swapping is used to move all of a process from main memory to disk. When all the process b putting it in the suspended state and transferring it to disk.

Reasons for process suspension

- Swapping
- Timing
- Interactive user request
- Parent process request

Swapping

OS needs to release required main memory to bring in a process that is ready to execute.

Timing

Process may be suspended while waiting for the next time interval.

Interactive user request

Process may be suspended for debugging purpose by user.

Parent process request

To modify the suspended process or to coordinate the activity of various descendants

PROCESS CONTROL BLOCK (PCB)

While creating a process the operating system performs several operations. To identify these processes, it must identify each process; hence it assigns a process identification number (PID) to each process. As the operating system supports multi-programming, it needs to keep track of all the processes. For this task, the process control block (PCB) is used to track the process's execution status. Each block of memory contains information about the process state, program counter, stack pointer, status of opened files, scheduling algorithms, etc. All these information is required and must be saved when the process is switched from one state to another. When the process made transitions from one state to another, the operating system must update information in the process's PCB.

Role of Process Control Block

The role or work of process control block (PCB) in process management is that it can access or modified by most OS utilities including those are involved with memory, scheduling, and input/output resource access. It can be said that the set of the process control blocks give the information of the current state of the operating system. Data structuring for processes is often done in terms of process control blocks. For example, pointers to other process control blocks inside any process control block allows the creation of those queues of processes in various scheduling states. The various information that is contained by process control block is listed below:

- Naming the process
- State of the process
- Resources allocated to the process
- Memory allocated to the process
- Scheduling information
- Input / output devices associated with process

COMPONENTS OF PCB

A Process Control Block is a data structure maintained by the Operating System for every process. The PCB is identified by an integer process ID (PID). A PCB keeps all the information needed to keep track of a process as listed below:

a. **Process State:** As we know that the process state of any process can be New, running, waiting, executing, blocked, suspended, terminated. For more details regarding process states you can refer process management of an Operating System. Process control block is used to define the process state of any process. In other words, process control block refers the states of the processes.

b. **Process privileges:** This is required to allow/disallow access to system resources.

c. **Process ID:** In computer system there are various processes running simultaneously and each process has its unique ID. This Id helps system in scheduling the processes. This Id is provided by the process control block. In other words, it is an identification number that uniquely identifies the processes of computer system.

d. **Pointer:** A pointer to parent process.

e. **Program Counter:** Program Counter is a pointer to the address of the next instructio to be executed for this process.

f. **CPU registers:** This information is comprising with the various registers, such as inde and stack that are associated with the process. This information is also managed by th process control block.

g. **CPU Scheduling Information:** Scheduling information is used to set the priority o different processes. This is very useful information which is set by the process contro block. In computer system there were many processes running simultaneously and each process have its priority. The priority of primary feature of RAM is higher than other secondary features. Scheduling information is very useful in managing any computer system.

h. **Memory management information:** This includes the information of page table memory limits, Segment table depending on memory used by the operating system.

i. **Accounting information:** This includes the amount of CPU used for process executio time limits, execution ID etc.

j. **IO status information:** This includes a list of I/O devices allocated to the process.

Since PCB contains the critical information for the process, it must be kept in an area of memory protected from normal user access. In some operating systems the PCB is placed in the beginning o the kernel stack of the process as it is a convenient protected location. The architecture of a PCB i completely dependent on Operating System and may contain different information in differen operating systems. The PCB is maintained for a process throughout its lifetime, and is deleted ono the process terminates. Here is a simplified diagram of a PCB;

Process ID
State
Pointer
Priority
Program Counter
CPU registers
I/O Information
Account Information
Etc....

Fig 3.3: PCB diagram

OPERATIONS ON PROCESS

There are many operations that can be performed on processes. Some of these are process creatio process preemption, process blocking, and process termination. These are given in detail as follows

Process Creation

Processes need to be created in the system for different operations. This can be done by the followi events:

- User request for process creation
- System initialization
- Execution of a process creation system call by a running process
- Batch job initialization

A process may be created by another process using fork(). The creating process is called the parent process and the created process is the child process. A child process can have only one parent but a parent process may have many children. Both the parent and child processes have the same memory image, open files, and environment strings. However, they have distinct address spaces.

A diagram that demonstrates process creation using fork() is as follows:

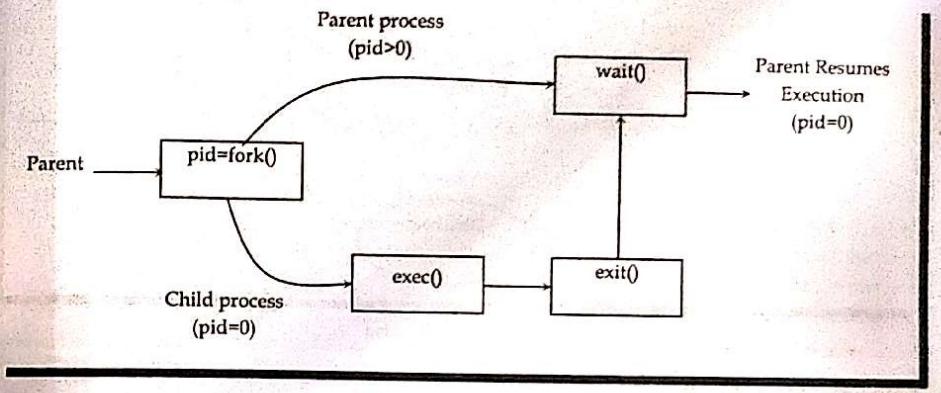


Fig 3.4 Process creation using fork()

Process Preemption

An interrupt mechanism is used in preemption that suspends the process executing currently and the next process to execute is determined by the short-term scheduler. Preemption makes sure that all processes get some CPU time for execution. A diagram that demonstrates process preemption is as follows:

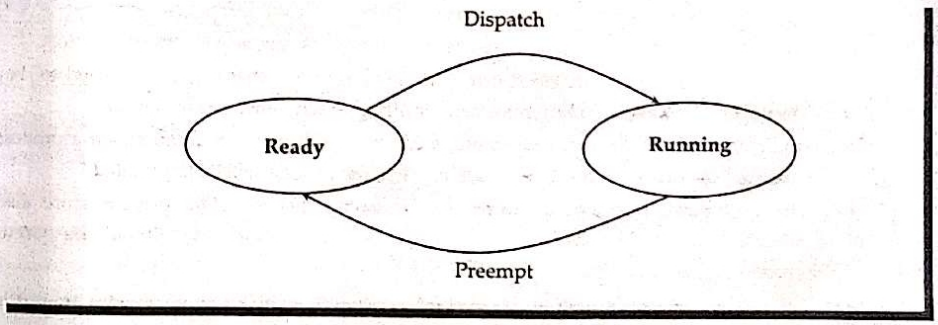


Fig 3.5: Process Preemption

Process Blocking

The process is blocked if it is waiting for some event to occur. This event may be I/O as the I/O events are executed in the main memory and don't require the processor. After the event is complete the process again goes to the ready state. A diagram that demonstrates process blocking is as follows

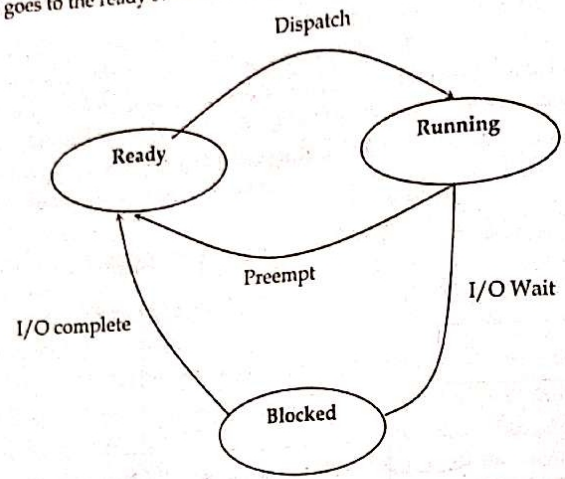


Fig 3.6: Process Blocking

Process Termination

After the process has completed the execution of its last instruction, it is terminated. The resources held by a process are released after it is terminated. A child process can be terminated by its parent process if its task is no longer relevant. The child process sends its status information to the parent process before it terminates. Also, when a parent process is terminated, its child processes are terminated as well as the child processes cannot run if the parent processes are terminated. Some of the causes of process termination are as follows:

- A process may be terminated after its execution is naturally completed. This process leaves the processor and releases all its resources.
- A child process may be terminated if its parent process requests for its termination.
- A process can be terminated if it tries to use a resource that it is not allowed to. For example - A process can be terminated for trying to write into a read only file.
- If an I/O failure occurs for a process, it can be terminated. For example - If a process requires the printer and it is not working, then the process will be terminated.
- In most cases, if a parent process is terminated then its child processes are also terminated. This is done because the child process cannot exist without the parent process.
- If a process requires more memory than is currently available in the system, then it is terminated because of memory scarcity.

PROCESS HIERARCHIES

The process of creating new process form their parent process and again creating new process from previous newly created p.ocess in tree like structure is called process hierarchy. Windows has no concept of process hierarchy.

Modern general purpose operating systems permit a user to create and destroy processes. In UNIX this is done by the fork system call, which creates a child process, and the exit system call, which terminates the current process. After a fork both parent and child keep running (indeed they have the same program text) and each can fork off other processes. The root of the tree is a special process created by the OS during startup.

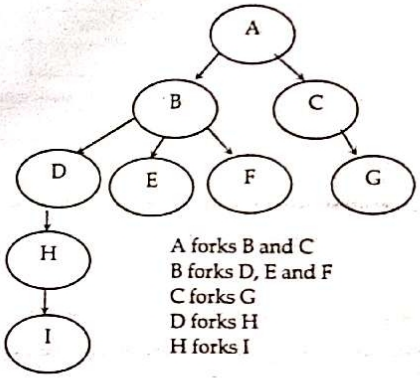


Fig 3.7: Process Hierarchy

PROCESS IMPLEMENTATION

Process Model is implemented by Process Table and Process Control Block which keep track all information of process. At the time of creation of a new process, operating system allocates a memory for it loads a process code in the allocated memory and setup data space for it. The state of process is stored as ' new ' in its PCB and when this process move to ready state its state is also changes in PCB. When a running process needs to wait for an input output devices, its state is changed to 'blocked'. The various queues used for this which is implemented as linked list.

DEFINITIONS OF THREADS

Despite of the fact that a thread must execute in process, the process and its associated threads a different concept. Processes are used to group resources together and threads are the entit scheduled for execution on the CPU.

A thread is a single sequence stream within in a process. Because threads have some of th properties of processes, they are sometimes called lightweight processes. In a process, threads allo multiple executions of streams. In many respect, threads are popular way to improve applicatio through parallelism. The CPU switches rapidly back and forth among the threads giving illusion th the threads are running in parallel. Like a traditional process i.e., process with one thread, a thre can be in any of several states (Running, Blocked, Ready or terminated). Each thread has its ow stack. Since thread will generally call different procedures and thus a different execution histor This is why thread needs its own stack. An operating system that has thread facility, the basic unit CPU utilization is a thread. A thread has or consists of a program counter (PC), a register set, and stack space. Threads are not independent of one other like processes as a result threads shares wi other threads their code section, data section, OS resources also known as task, such as open files an signals.

Traditional (heavyweight) processes have a single thread of control - There is one program counte and one sequence of instructions that can be carried out at any given time, as shown in figure belo multi-threaded applications have multiple threads within a single process, each having their ow program counter, stack and set of registers, but sharing common code, data, and certain structure such as open files.

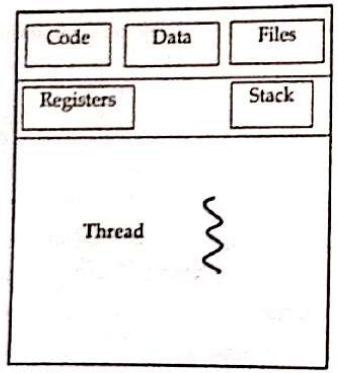


Fig 3.12: Single threaded process

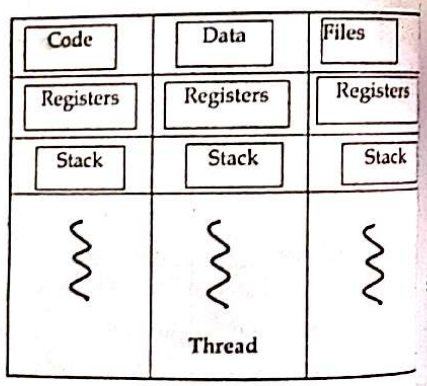


Fig 3.13: Multi threaded process

Threads are very useful in modern programming whenever a process has multiple tasks to perfor independently of the others. This is particularly true when one of the tasks may block, and it desired to allow the other tasks to proceed without blocking. For example, in a word processor background thread may check spelling and grammar while a foreground thread processes user inp (keystrokes), while yet a third thread loads images from the hard drive, and a fourth does period automatic backups of the file being edited.

Another example is a web server - Multiple threads allow for multiple requests to be satisfied simultaneously, without having to service requests sequentially or to fork off separate processes for every incoming request. (The latter is how this sort of thing was done before the concept of threads was developed. A daemon would listen at a port, fork off a child for every incoming request to be processed, and then go back to listening to the port). There are four major benefits to multi-threading:

- **Responsiveness** - One thread may provide rapid response while other threads are blocked or slowed down doing intensive calculations.
- **Resources haring** - By default threads share common code, data, and other resources, which allows multiple tasks to be performed simultaneously in a single address space.
- **Economy** - Creating and managing threads (and context switches between them) is much faster than performing the same tasks for processes.
- **Scalability, i.e. Utilization of multiprocessor architectures** - A single threaded process can only run on one CPU, no matter how many may be available, whereas the execution of a multi-threaded application may be split amongst available processors. (Note that single threaded processes can still benefit from multi-processor architectures when there are multiple processes contending for the CPU, i.e. when the load average is above some certain threshold.)

Properties of a Thread

- Only one system call can create more than one thread (Lightweight process).
- Threads share data and information.
- Threads shares instruction, global and heap regions but have its own individual stack and registers.
- Thread management consumes no or fewer system calls as the communication between threads can be achieved using shared memory.
- The isolation property of the process increases its overhead in terms of resource consumption.

THE THREAD MODEL

There are two types of threads to be managed in a modern system: User threads and kernel threads. User threads are supported above the kernel, without kernel support. These are the threads that application programmers would put into their programs. Kernel threads are supported within the kernel of the OS itself. All modern operating systems support kernel level threads, allowing the kernel to perform multiple simultaneous tasks and/or to service multiple kernel system calls simultaneously. In a specific implementation, the user threads must be mapped to kernel threads, using one of the following strategies.

- **Many-To-One Model:** This model maps many user level threads to one kernel level thread. Thread management is done by thread library in user space, so it is efficient. However, if a blocking system call is made, then the entire process blocks, even if the

other user threads would otherwise be able to continue. Because a single kernel thread can operate only on a single CPU, the many-to-one model does not allow individual processes to be split across multiple CPUs. Green threads for Solaris and GNU Portable Threads implement the many-to-one model in the past, but few systems continue to do so today.

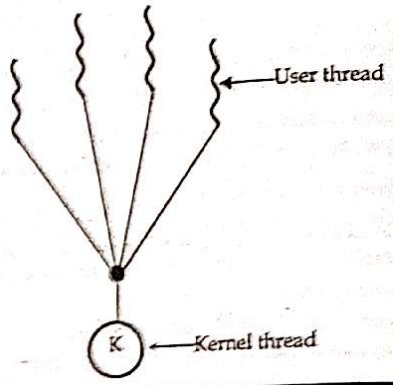


Fig 3.14: many to one thread model

- **One-To-One Model:** One user thread is mapped to one kernel thread. It provides more concurrency than previous model by allowing another thread to run during blocking. It also allows parallelism. The only overhead is for each threads corresponding kernel thread should be created. Most implementations of this model place a limit on how many threads can be created. Linux and Windows from 95 to XP implement the one-to-one model for threads.

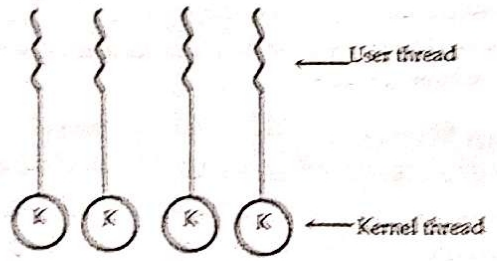


Fig 3.15: one to one thread model

- **Many-To-Many Model:** In this model, many user level threads multiplex to the Kernel thread of smaller or equal numbers. The number of Kernel threads may be specific to either a particular application or a particular machine. Users have no restrictions on the number of threads created. Blocking kernel system calls do not block the entire process. Processes can be split across multiple processors. Individual processes may be allocated variable numbers of kernel threads, depending on the number of CPUs present and other factors.

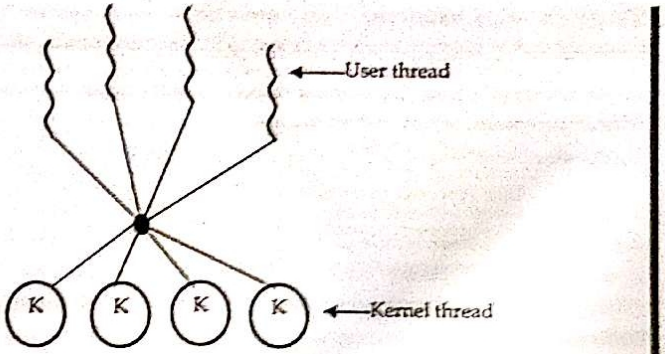


Fig 3.16: Many-to-many thread model

IMPLEMENTING THREADS IN USER SPACE

The first method is to put the threads package entirely in user space. The kernel knows nothing about them. As far as the kernel is concerned, it is managing ordinary, single-threaded processes. The first, and most obvious, advantage is that a user-level threads package can be implemented on an operating system that does not support threads. All operating systems used to fall into this category, and even now some still do.

The threads run on top of a run-time system, which is a collection of procedures that manage threads. When threads are managed in user space, each process needs its own private thread table. It is analogous to the kernel's process table, except that it keeps track only of the per-thread properties such each thread's program counter stack pointer, registers, state, etc. The thread table is managed by the runtime system. When a thread is moved to ready state or blocked state, the information needed to restart it is stored in the thread table, exactly the same way as the kernel stores information about processes in the process table.

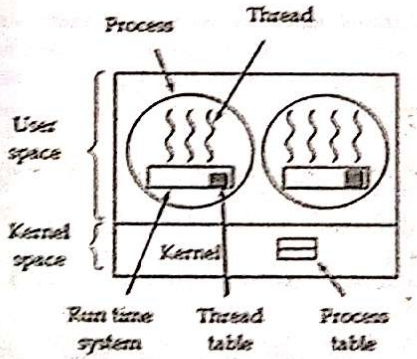


Fig 3.17: A user-level threads package.

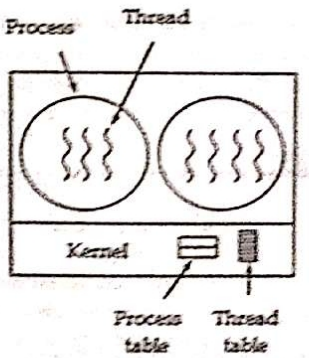


Fig 3.18: A threads package managed by the kernel.

THREAD VS. PROCESS

As we mentioned earlier that in many respect threads operate in the same way as that of proces
Some of the similarities and differences are:

Similarities

- Like processes threads share CPU and only one thread active (running) at a time.
- Like processes, threads within a process, threads within a process execute sequentiall
- Like processes, thread can create children.
- And like process, if one thread is blocked, another thread can run.

Differences

- All threads of a program are logically contained within a process.
- A process is heavy weighted, but a thread is light weighted.
- A program is an isolated execution unit whereas thread is not isolated and sha memory.
- A thread cannot have an individual existence; it is attached to a process. On the oth hand, a process can exist individually.
- At the time of expiration of a thread, its associated stack could be recovered as eve thread has its own stack. In contrast, if a process dies, all threads die including t process.

KERNEL-LEVEL THREADS

To make concurrency cheaper, the execution aspect of process is separated out into threads. As su the OS now manages threads and processes. All thread operations are implemented in the ker and the OS schedules all threads in the system. OS managed threads are called kernel-level thre or light weight processes.

In this method, the kernel knows about and manages the threads. No runtime system is needed this case. Instead of thread table in each process, the kernel has a thread table that keeps track of threads in the system. In addition, the kernel also maintains the traditional process table to ke track of processes. Operating Systems kernel provides system call to create and manage threads.

Advantages

- Because kernel has full knowledge of all threads, Scheduler may decide to give m time to a process having large number of threads than process having small numbe threads.
- Kernel-level threads are especially good for applications that frequently block.

Disadvantages

- The kernel-level threads are slow and inefficient. For instance, threads operations hundreds of times slower than that of user-level threads.

- Since kernel must manage and schedule threads as well as processes. It requires a full thread control block (TCB) for each thread to maintain information about threads. As a result, there is significant overhead and increased in kernel complexity.

USER-LEVEL THREADS

Kernel-Level threads make concurrency much cheaper than process because, much less state to allocate and initialize. However, for fine-grained concurrency, kernel-level threads still suffer from too much overhead. Thread operations still require system calls. Ideally, we require thread operations to be as fast as a procedure call. Kernel-Level threads have to be general to support the needs of all programmers, languages, runtimes, etc. For such fine grained concurrency, we need still cheaper threads.

To make threads cheap and fast, they need to be implemented at user level. User-Level threads are managed entirely by the run-time system (user-level library). The kernel knows nothing about user-level threads and manages them as if they were single-threaded processes. User-Level threads are small and fast, each thread is represented by a PC, register, stack, and small thread control block. Creating a new thread, switching between threads, and synchronizing threads are done via procedure call i.e. no kernel involvement. User-Level threads are hundred times faster than Kernel-Level threads.

Advantages

The most obvious advantage of this technique is that a user-level threads package can be implemented on an Operating System that does not support threads. Some other advantages are

- User-level threads do not require modification to operating systems.
- Simple Representation: Each thread is represented simply by a PC, registers, stack and a small control block, all stored in the user process address space.
- Simple Management: This simply means that creating a thread, switching between threads and synchronization between threads can all be done without intervention of the kernel.
- Fast and Efficient: Thread switching is not much more expensive than a procedure call.

Disadvantages

- There is a lack of coordination between threads and operating system kernel. Therefore, process as whole gets one time slice irrespective of whether process has one thread or 1000 threads within. It is up to each thread to relinquish control to other threads.
- User-level threads require non-blocking systems call i.e., a multithreaded kernel. Otherwise, entire process will have blocked in the kernel, even if there are runnable threads left in the processes. For example, if one thread causes a page fault, the process blocks.

INTER PROCESS COMMUNICATION

Inter-process communication (IPC) is a mechanism that allows the exchange of data betw processes. By providing a user with a set of programming interfaces, IPC helps a program organize the activities among different processes. IPC allows one application to control and application, thereby enabling data sharing without interference. IPC enables data communicatio allowing processes to use segments, semaphores, and other methods to share memory a information. IPC facilitates efficient message transfer between processes. The idea of IPC is based Task Control Architecture (TCA). It is a flexible technique that can send and receive variable leng arrays, data structures, and lists. It has the capability of using publish/subscribe and client/ser data-transfer paradigms while supporting a wide range of operating systems and languages.

Working together with multiple processes, require an inter-process communication (IPC) meth which will allow them to exchange data along with various information. There are two prima models of inter-process communication:

- shared memory and
- Message passing.

In the shared-memory model, a region of memory which is shared by cooperating processes is established. Processes can be then able to exchange information by reading and writing all the da to the shared region. In the message-passing form, communication takes place by way of messa exchanged among the cooperating processes. The two communications models are contrasted in t figure below:

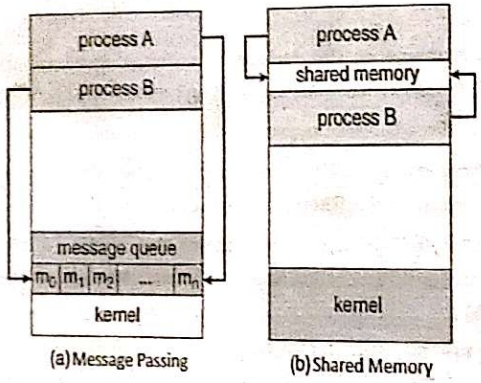


Fig 3.19: Inter process communication

RACE CONDITION

A race condition occurs when two or more threads can access shared data and they try to change the same time. Because the thread scheduling algorithm can swap between threads at any time, don't know the order in which the threads will attempt to access the shared data. Therefore, result of the change in data is dependent on the thread scheduling algorithm, i.e. both threads racing to access/change the data.

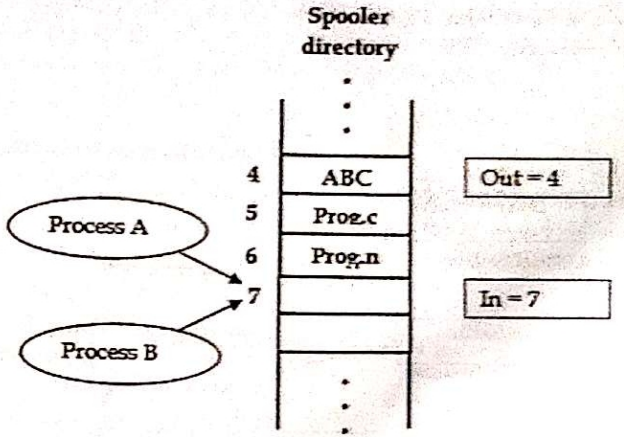


Fig 3.20: IPC with race condition

A print spooler: When a process wants to print a file it enters the filename in a special spooler directly. Another process, the Printer Daemon, periodically checks to see if there are any files to be printed, and if there are, it prints them and removes their name from the directory. Imagine that our spooler directory has a large number of slots, numbered 0, 1, 2 ... each one capable of holding a filename. Also imagine that there are two shared variables,

- Out: points to next file to printed.
- In: points to next free slot in the directory.

Slots 0 to 3 files already printed. Slots 4 to 6 files names which has to be printed.

Now the main issue comes:

Process A reads in and store the value, 7, in a local variable called next-free-slot. Just then a clock interrupt occurs and the CPU decides that process A has run long enough, it switches to process B.

Process B also reads in, and also gets a 7, so it stores the name of its in slot 7 and update into 8. Then it goes off and does other things.

Eventually, process A once again, starting from the place at left off last time. It looks next free slot, finds a7 there, and writes its file name 7 in slot 7, erasing the name that process B just put there. Then it computes next-free-slot+1, which is 8, and sets in to 8.

The spooler directory is now internally consistent, so the printer daemon will not notice anything wrong, but process B will never receive any output.

CRITICAL SECTION

How do we avoid race conditions? One way to prevent two or more process, using the shared data is mutual exclusion i.e. some way of making sure that if one process is using a shared variable or file, the process will be excluded from doing same thing. Sometimes a process has to access shared memory or files that can lead to races. That part of the program where the shared memory is accessed is called the critical region or critical section. If we could arrange matters such that no two

processes were ever in their critical regions at the same time, we could avoid races. The
section is given as follows:

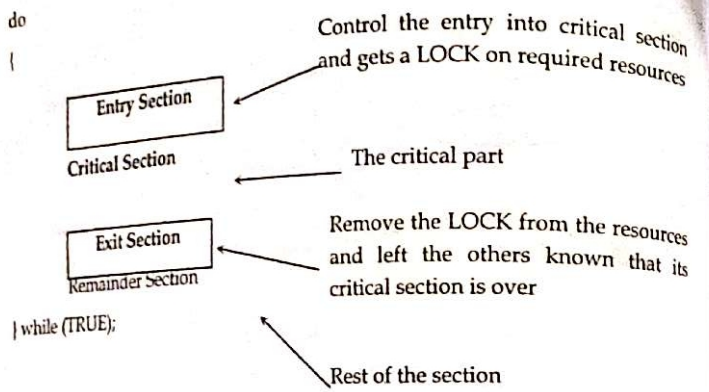


Fig 3.21: Critical section

In the above diagram, the entry sections handle the entry into the critical section. It acquires resources needed for execution by the process. The exit section handles the exit from the cri section. It releases the resources and also informs the other processes that critical section is free.

The critical section problem needs a solution to synchronize the different processes. The solutio the critical section problem must satisfy the following conditions:

a. **Mutual Exclusion:** Mutual exclusion implies that only one process can be inside t critical section at any time. If any other processes require the critical section, they m wait until it is free.

b. **Progress:** Progress means that if a process is not using the critical section, then it sho not stop any other process from accessing it. In other words, any process can ente critical section if it is free.

c. **Bounded Waiting:** Bounded waiting means that each process must have a limit waiting time. It should not wait endlessly to access the critical section.

AVOIDING CRITICAL REGION

To avoiding critical region we need process synchronization. Process Synchronization is the task coordinating the execution of processes in a way that no two processes can have access to the sa shared data and resources. It is specially needed in a multi-process system when multiple process are running together, and more than one process try to gain access to the same shared resource data at the same time.

This can lead to the inconsistency of shared data. So the change made by one process not necessarily reflected when other processes accessed the same shared data. To avoid this type of inconsistency of data, the processes need to be synchronized with each other. Mutual exclusion and serializability is the best way of avoiding critical regions.

Mutual Exclusion and Serializability

Mutual exclusion is a concurrency control property which is introduced to prevent race conditions. It is the requirement that a process cannot enter its critical section while another concurrent process is currently present or executing in its critical section i.e. only one process is allowed to execute the critical section at any given instance of time.

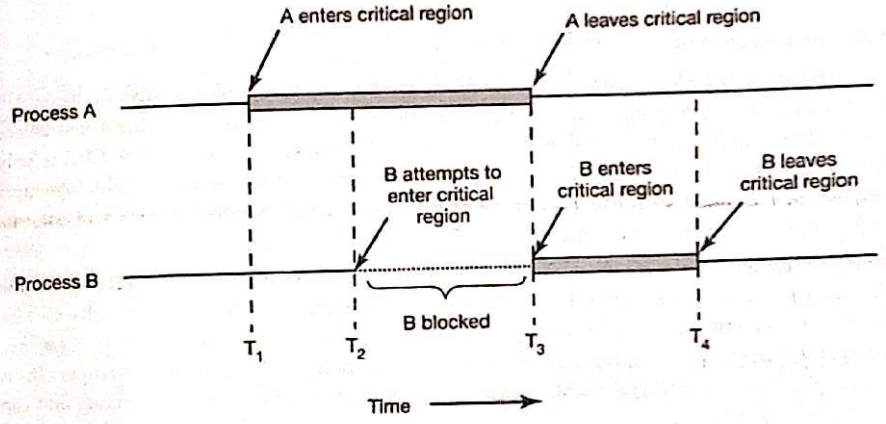


Fig 3.22: Mutual exclusion

At time T1 process A enters its critical region. At T2 process B attempts to enter its critical region but fails. Until T3, process B is temporarily suspended. At time T3, B enters its critical region. At T4, B leaves its critical region.

MUTUAL EXCLUSION CONDITIONS

If we could arrange matters such that no two processes were ever in their critical sections simultaneously, we could avoid race conditions. We need four conditions to hold to have a good solution for the critical section problem (mutual exclusion).

- No two processes may at the same moment inside their critical sections.
- No assumptions are made about relative speeds of processes or number of CPUs.
- No process should outside its critical section should block other processes.
- No process should wait arbitrary long to enter its critical section.

PROCESS SCHEDULING

The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy. Process scheduling is an essential part of multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

Process Scheduling Queues

The OS maintains all PCBs in Process Scheduling Queues. The OS maintains a separate queue for each of the process states and PCBs of all processes in the same execution state are placed in the same queue. When the state of a process is changed, its PCB is unlinked from its current queue and moved to its new state queue.

The Operating System maintains the following important process scheduling queues –

- **Job queue** – This queue keeps all the processes in the system.
- **Ready queue** – This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.
- **Device queues** – The processes which are blocked due to unavailability of an I/O device constitute this queue.

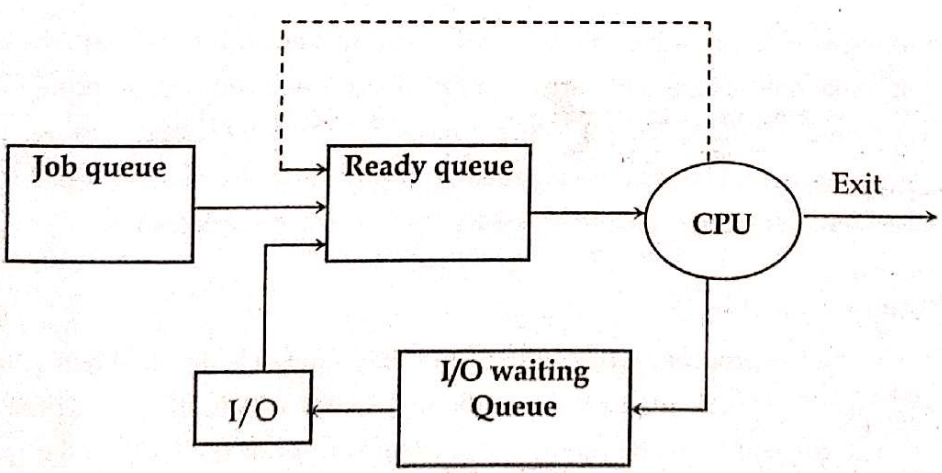


Fig 3.31: Process scheduling queue

The OS can use different policies to manage each queue (FIFO, Round Robin, Priority, etc.). The OS scheduler determines how to move processes between the ready and run queues which can only have one entry per processor core on the system; in the above diagram, it has been merged with the CPU.

Schedulers

Schedulers are special system software which handles process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run. Schedulers are of three types

a. **Long Term Scheduler:** It is also called a job scheduler. A long-term scheduler d... which programs are admitted to the system for processing. It selects processes from ... and loads them into memory for execution. Process loads into the memory ... scheduling.

The primary objective of the job scheduler is to provide a balanced mix of jobs, such ... bound and processor bound. It also controls the degree of multiprogramming. If the ... multiprogramming is stable, then the average rate of process creation must be equal ... average departure rate of processes leaving the system. On some systems, the long ... scheduler may not be available or minimal. Time-sharing operating systems have ... term scheduler. When a process changes the state from new to ready, then there is use ... term scheduler.

b. **Short Term Scheduler:** It is also called as CPU scheduler. Its main objective is to ... system performance in accordance with the chosen set of criteria. It is the change of ... state to running state of the process. CPU scheduler selects a process among the process ... are ready to execute and allocates CPU to one of them. Short-term schedulers, also kn... dispatchers, make the decision of which process to execute next. Short-term schedule... faster than long-term schedulers.

c. **Medium Term Scheduler:** Medium-term scheduling is a part of swapping. It remov... processes from the memory. It reduces the degree of multiprogramming. The mediu... scheduler is in-charge of handling the swapped out-processes.

A running process may become suspended if it makes an I/O request. A suspended p... cannot make any progress towards completion. In this condition, to remove the proces... memory and make space for other processes, the suspended process is moved t... secondary storage. This process is called swapping, and the process is said to be swapped ... or rolled out. Swapping may be necessary to improve the process mix.

Dispatcher

Dispatcher is a special program which comes into play after scheduler. When scheduler compl... its job of selecting a process, then after it is the dispatcher which takes that process to the ... state/queue. The dispatcher is the module that gives control of the CPU to the process selec... the short-term scheduler. This function involves the following:

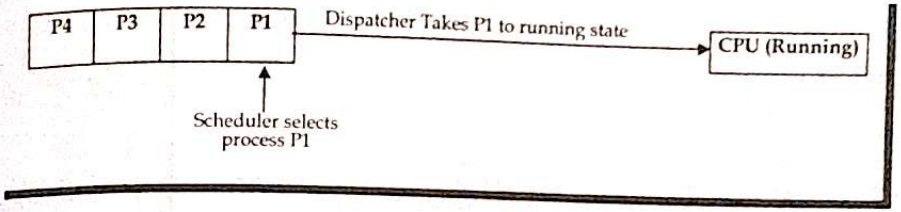
- Switching context
- Switching to user mode
- Jumping to proper location in user program to restart that program

Difference between the Scheduler and Dispatcher

Consider a situation, where various process residing in ready queue and waiting for execution ... CPU can't execute all the process of ready queue simultaneously, operating system have to cho... particular process on the basis of scheduling algorithm used. So, this procedure of selecting a pr... among various processes is done by scheduler. Now here the task of scheduler completed ...

dispatcher comes into picture as scheduler have decide a process for execution, it is dispatcher who takes that process from ready queue to the running status, or you can say that providing CPU to that process is the task of dispatcher.

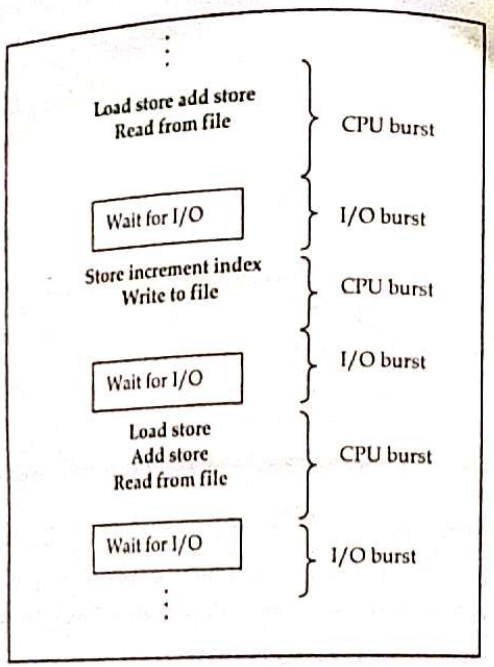
Example: There are 4 processes in ready queue, i.e., P1, P2, P3, P4; they all are arrived at t0, t1, t2, t3 respectively. First in First out scheduling algorithm is used. So, scheduler decided that first of all P1 has came, so this is to be executed first. Now dispatcher takes P1 to the running state.



Dispatcher	Scheduler
Dispatcher is a module that gives control of CPU to the process selected by short term scheduler.	Scheduler is something which selects a process among various processes.
There are no different types in dispatcher. It is just a code segment.	There are 3 types of scheduler i.e. Long-term, Short-term, Medium-term.
Working of dispatcher is dependent on scheduler. Means dispatchers have to wait until scheduler selects a process.	Scheduler works independently. It works immediately when needed.
Dispatcher has no specific algorithm for its implementation	Scheduler works on various algorithm such as FCFS, SJF, RR etc.
The time taken by dispatcher is called dispatch latency.	Time taken by scheduler is usually negligible. Hence we neglect it.
Dispatcher is also responsible for: Context Switching, Switch to user mode, Jumping to proper location when process again restarted	The only work of scheduler is selection of processes.

CPU - I/O Burst Cycle

The success of CPU scheduling depends on the following observed property of processes: Process execution consists of a cycle of CPU execution and I/O wait. Processes alternate back and forth between these two states. The execution begins with CPU burst, followed by I/O burst, then another CPU burst and so on. The last CPU burst will end with a system request to terminate execution rather than with another I/O burst. An I/O bound program would typically have many short CPU bursts; a CPU bound program might have a few very long CPU bursts. The duration of these CPU bursts are measured, which help to select an appropriate CPU scheduling algorithm.



Context Switch

A context switch is the mechanism to store and restore the state or context of a CPU in h
Control block so that a process execution can be resumed from the same point at a later time
this technique, a context switcher enables multiple processes to share a single CPU. C
switching is an essential part of a multitasking operating system features.

When the scheduler switches the CPU from executing one process to execute another, the sta
the current running process is stored into the process control block. After this, the state f
process to run next is loaded from its own PCB and used to set the PC, registers, etc. At that
the second process can start executing. Context switches are computationally intensive since
and memory state must be saved and restored. To avoid the amount of context switching time
hardware systems employ two or more sets of processor registers. When the process is switche
following information is stored for later use.

- Program Counter
- Scheduling information
- Base and limit register value
- Currently used register
- Changed State
- I/O State information
- Accounting information

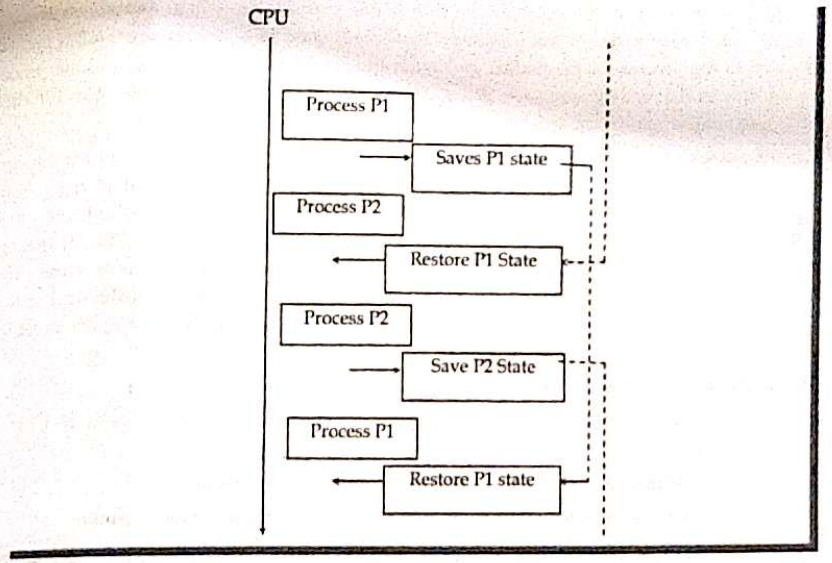


Fig 3.32: Context switching

TYPES OF SCHEDULING

An operating system uses two types of scheduling processes execution, preemptive and non - preemptive. CPU scheduling decisions may take place under the following four circumstances:

1. When a process switches from the running state to the waiting state (for I/O request or invocation of wait for the termination of one of the child processes).
2. When a process switches from the running state to the ready state (for example, when an interrupt occurs).
3. When a process switches from the waiting state to the ready state(for example, completion of I/O).
4. When a process terminates.

In circumstances 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution. There is a choice, however in circumstances 2 and 3.

When Scheduling takes place only under circumstances 1 and 4, we say the scheduling scheme is non-preemptive; otherwise the scheduling scheme is preemptive.

Preemptive Scheduling

In this type of Scheduling, the tasks are usually assigned with priorities. At times it is necessary to run a certain task that has a higher priority before another task although it is running. Therefore, the running task is interrupted for some time and resumed later when the priority task has finished its execution.

Preemptive scheduling is one which can be done in the circumstances when a process switches from running state to ready state or from waiting state to ready state. Here, the resources (CPU cycles) are allocated to the process for the limited amount of time and then is taken away, and the process is placed back in the ready queue again if it still has CPU burst time remaining. The process stays in ready queue till it gets next chance to execute.

If a process with high priority arrives in the ready queue, it does not have to wait for the current process to complete its burst time. Instead, the current process is interrupted in the middle of execution and is placed in the ready queue till the process with high priority is utilizing the CPU cycles. In this way, each process in the ready queue gets some time to run CPU. It makes preemptive scheduling flexible but increases the overhead of switching the process from running state to ready state and vise-verse. Algorithms that work on preemptive scheduling are Round Robin, Shortest Job First (SJF) and Priority scheduling may or may not come under preemptive scheduling.

Characteristics of Preemptive Scheduling

- Picks a process and let it run for a maximum of fixed time and releases the CPU after that quantum, whether it finishes or not.
- Processes are allowed to run for a maximum of some fixed time.
- Useful in systems in which high-priority processes requires rapid attention.
- In time sharing systems, preemptive scheduling is important in guaranteeing acceptable response times.
- High overhead.

Non-preemptive Scheduling

Under non-preemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state.

Non-preemptive Scheduling is one which can be applied in the circumstances when a process terminates, or a process switches from running to waiting state. In Non-Preemptive Scheduling, once the resources (CPU) are allocated to a process, the process holds the CPU till it gets terminated or reaches a waiting state. Unlike preemptive scheduling, non-preemptive scheduling does not interrupt a process running CPU in middle of the execution. Instead, it waits for the process to complete its CPU burst time and then it can allocate the CPU to another process. In Non-preemptive scheduling, if a process with long CPU burst time is executing then the other process will have to wait for a long time which increases the average waiting time of the processes in the ready queue. However, the non-preemptive scheduling does not have any overhead of switching the process from ready queue to CPU but it makes the scheduling rigid as the process in execution is not even preempted for a process with higher priority.

Characteristics of Non-Preemptive Scheduling

- Picks a process to run until it releases the CPU
- Once a process has been given the CPU, it runs until blocks for I/O or termination
- Treatment of all processes is fair
- Response times are more predictable
- Useful in real-time system
- Shorts jobs are made to wait by longer jobs -no priority

Comparisons of preemptive and non-preemptive scheduling

Preemptive scheduling	Non preemptive scheduling
The resources are allocated to a process for a limited time.	Once resources are allocated to a process, the process holds it till it completes its burst time or switches to waiting state.
Process can be interrupted in between.	Process cannot be interrupted till it terminates or switches to waiting state.
If a high priority process frequently arrives in the ready queue, low priority process may starve.	If a process with long burst time is running CPU, then another process with less CPU burst time may starve.
Preemptive scheduling has overheads of scheduling the processes.	Non-preemptive scheduling does not have overheads.
Preemptive scheduling is flexible.	Non-preemptive scheduling is rigid.
Preemptive scheduling is cost associated.	Non-preemptive scheduling is not cost associative.

SCHEDULING CRITERIA

Different CPU scheduling algorithms have different properties and may favor one class of processes over another. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms. Many criteria have been suggested for comparing CPU scheduling algorithms. Criteria that are used include the following:

1. **CPU utilization** – keep the CPU as busy as possible
2. **Throughput** – number of processes that complete their execution per time unit
3. **Turnaround time** – amount of time to execute a particular process
4. **Waiting time** – amount of time a process has been waiting in the ready queue
5. **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

The goals of scheduling are as follows:

- **Fairness:** Each process gets fair share of the CPU.
- **Efficiency:** When CPU is 100% busy then efficiency is increased.
- **Response Time:** Minimize the response time for interactive user.
- **Throughput:** Maximizes jobs per given time period.
- **Waiting Time:** Minimizes total time spent waiting in the ready queue.
- **Turnaround Time:** Minimizes the time between submission and termination.

SCHEDULING ALGORITHM

Scheduling algorithms mainly divided into following three categories

- Batch system scheduling
- Interactive system scheduling and
- Real Time Scheduling

Batch System Scheduling

Batch processing is a technique in which an Operating System collects the programs and data together in a batch before processing starts. An operating system does the following activities related to batch processing:

- The OS defines a job which has predefined sequence of commands, programs and data as a single unit.
- The OS keeps a number a job in memory and executes them without any manual information.
- Jobs are processed in the order of submission, i.e., first come first served fashion.
- When a job completes its execution, its memory is released and the output for the job gets copied into an output spool for later printing or processing.

The some major terminologies used in various scheduling algorithms are:

- **Arrival Time:** Time at which the process arrives in the ready queue.
- **Completion Time:** Time at which process completes its execution.
- **Burst Time:** Time required by a process for CPU execution.
- **Turn Around Time:** Time Difference between completion time and arrival time.

  Turn Around Time = Completion Time – Arrival Time
- **Waiting Time (W.T):** Time Difference between turnaround time and burst time.

  Waiting Time = Turn Around Time – Burst Time

The major process scheduling algorithms based on batch system scheduling are described as below;

a. **First come first served**

FCFS provides an efficient, simple and error-free process scheduling algorithm that saves valuable CPU resources. It uses non-preemptive scheduling in which a process is automatically queued and processing occurs according to an incoming request or process order. FCFS derives its concept from real-life customer service.

Let's take a look at how FCFS process scheduling works. Suppose there are three processes in a queue: P1, P2 and P3. P1 is placed in the processing register with a waiting time of zero seconds and 10 seconds for complete processing. The next process, P2, must wait 10 seconds and is placed in the processing cycle until P1 is processed. Assuming that P2 will take 15 seconds to complete, the final process, P3, must wait 25 seconds to be processed. FCFS may not be the fastest process scheduling algorithm, as it does not check for priorities associated with processes. These priorities may depend on the processes' individual execution times.

SCHEDULING ALGORITHM

Scheduling algorithms mainly divided into following three categories

- Batch system scheduling
- Interactive system scheduling and
- Real Time Scheduling

Batch System Scheduling

Batch processing is a technique in which an Operating System collects the programs and together in a batch before processing starts. An operating system does the following activities to batch processing:

- The OS defines a job which has predefined sequence of commands, programs and as a single unit.
- The OS keeps a number a job in memory and executes them without any information.
- Jobs are processed in the order of submission, i.e., first come first served fashion.
- When a job completes its execution, its memory is released and the output for th gets copied into an output spool for later printing or processing.

The some major terminologies used in various scheduling algorithms are:

- **Arrival Time:** Time at which the process arrives in the ready queue.
- **Completion Time:** Time at which process completes its execution.
- **Burst Time:** Time required by a process for CPU execution.
- **Turn Around Time:** Time Difference between completion time and arrival time.

  Turn Around Time = Completion Time – Arrival Time
- **Waiting Time (W.T):** Time Difference between turnaround time and burst time.

  Waiting Time = Turn Around Time – Burst Time

The major process scheduling algorithms based on batch system scheduling are described as belo

a. **First come first served**

FCFS provides an efficient, simple and error-free process scheduling algorithm that s valuable CPU resources. It uses non-preemptive scheduling in which a proces automatically queued and processing occurs according to an incoming request or pro order. FCFS derives its concept from real-life customer service.

Let's take a look at how FCFS process scheduling works. Suppose there are three process a queue: P1, P2 and P3. P1 is placed in the processing register with a waiting time of seconds and 10 seconds for complete processing. The next process, P2, must wait 10 seco and is placed in the processing cycle until P1 is processed. Assuming that P2 will tak seconds to complete, the final process, P3, must wait 25 seconds to be processed. FCFS not be the fastest process scheduling algorithm, as it does not check for priorities associ with processes. These priorities may depend on the processes' individual execution times

Characteristics

- Processes are scheduled in the order they are received.
- Once the process has the CPU, it runs to completion
- Easily implemented, by managing a simple queue or by storing time the process was received.
- Fair to all processes.

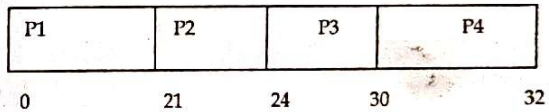
Problems

- No guarantee of good response time.
- Large average waiting time.

Example 1: Consider the processes P1, P2, P3, P4 given in the below table, arrives for execution in the order, with zero Arrival Time and given Burst Time, let's find the average waiting time and Average turnaround time using the FCFS scheduling algorithm.

Process	Burst Time
P1	21
P2	3
P3	6
P4	2

The Gnatt chart is as follows,



In the above example, we can see that we have four processes P1, P2, P3 and P4 and they are coming in the ready state at 0 ms. So the process P1 will be executed for the first 21 ms. After that, the process P2 will be executed for 3 ms and then process P3 will be executed for 6 ms and finally, the process P4 will be executed for 2 ms. One thing to be noted here is that if the arrival time of the processes is the same, then the CPU can select any process.

Process	Turnaround time = Completion Time – Arrival Time	Waiting Time = Turn Around Time – Burst Time
P1	21- 0=21 ms	21-21=0
P2	24 ms	24-3=21
P3	30 ms	30-6=24
P4	32 ms	32-2=30

Total waiting time: (0 + 21 + 24 + 30) = 75 ms

Average waiting time: (75/4) = 18.75 ms

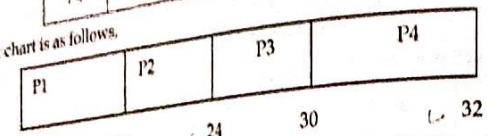
Total turnaround time: (21 + 24 + 30 + 32) = 107 ms

Average turnaround time: (107/4) = 26.75 ms

Example 2: Consider the processes P1, P2, P3, P4 given in the below table, arrives for execution in the order, with Arrival Time 0, 2, 2 and 3 respectively and given Burst Time, let's find the average waiting time and Average turnaround time using the FCFS scheduling algorithm.

Process	Arrival time	Burst Time
P1	0 ms	21
P2	2 ms	3
P3	2 ms	6
P4	3 ms	2

The Gnatt chart is as follows,



In the above example, we can see that we have four processes P1, P2, P3 and P4 and they are coming in the ready state at 0 ms, 2 ms, 2 ms and 3 ms respectively. So, based on the arrival time, the process P1 will be executed for the first 21 ms. After that, the process P2 will be executed for 3 ms and the process P3 will be executed for 6 ms and finally, the process P4 will be executed for 2 ms. One thing to be noted here is that if the arrival time of the processes is the same, then the CPU can select any process.

Process	Turnaround time = Completion Time – Arrival Time	Waiting Time = Turn Around Time – Burst Time
P1	21- 0=21 ms	21-21=0
P2	24-2=22 ms	22-3=19
P3	30-2=28 ms	28-6=22
P4	32-3=29 ms	29-2=27

Total waiting time: (0 + 19 + 22 + 27) = 68 ms

Average waiting time: (68/4) = 17 ms

Total turnaround time: (21 + 22 + 28 + 29) = 100 ms

Average turnaround time: (100/4) = 25 ms

b. Shortest job first

In the FCFS, we saw if a process is having a very high burst time and it comes first then all other process with a very low burst time have to wait for its turn. So, to remove this problem we come with a new approach i.e. Shortest Job First or SJF.

In this technique, the process having the minimum burst time at a particular instant of time will be executed first. If the subsequent CPU bursts of two processes become the same, then FCFS scheduling is used to break the tie. It is a non-preemptive approach i.e. if the process starts its execution then it will be fully executed and then some other process will come.

Shortest job first (SJF) or shortest job next, is a scheduling policy that selects the waiting process with the smallest execution time to execute next. Shortest Job first has the advantage of having minimum average waiting time among all scheduling algorithms.

Characteristics

- The processing times are known in advanced.
- SJF selects the process with shortest expected processing time. In case of the tie FCFS scheduling is used.
- The decision policies are based on the CPU burst time. Advantages:
- Reduces the average waiting time over FCFS.
- Favors shorts jobs at the cost of long jobs.

Problems

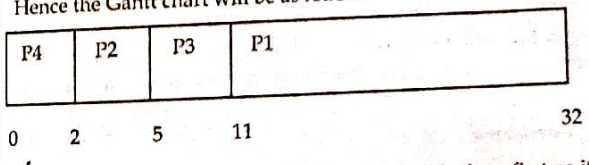
- It may lead to starvation if only short burst time processes are coming in the ready state.
- Estimation of run time to completion.
- Accuracy
- Not applicable in timesharing system.

Example 1: Consider the processes P1, P2, P3, P4 given in the below table, arrives for execution in the same order, with Arrival Time 0, and given Burst Time, let's find the average waiting time and Average turnaround time using the SJF scheduling algorithm.

Process	Burst Time
P1	21
P2	3
P3	6
P4	2

Solution: In shortest Job First Scheduling, the shortest Process is executed first.

Hence the Gantt chart will be as follow



As in the Gantt chart above, the process P4 will be picked up first as it has the shortest burst time, then P2, followed by P3 and at last P1.

Process	Turnaround time = Completion Time – Arrival Time	Waiting Time = Turn Around Time – Burst Time
P4	2- 0=2 ms	2-2=0
P2	5 ms	5-3=2
P3	11 ms	11-6=5
P1	32 ms	32-21=11

Total waiting time: (0 + 2 + 5 + 11) = 18 ms

Average waiting time: (18/4) = 4.50 ms

Total turnaround time: (2 + 5 + 11 + 32) = 50 ms

Average turnaround time: (50/4) = 12.50 ms

We scheduled the same set of processes using the First come first serve algorithm, and got average waiting time to be 18.75 ms, whereas with SJF, the average waiting time comes out 4.5 ms.

c. **Shortest remaining time next**

It is a preemptive version of shortest job next scheduling. In this scheduling algorithm, the process with the smallest amount of time remaining until completion is selected to execute. Since the currently executing process is the one with the shortest amount of time remaining by definition, and since that time should only reduce as execution progresses, processes will always run until they complete or a new process is added that requires a smaller amount of time.

Shortest remaining time is advantageous because short processes are handled very quickly. The system also requires very little overhead since it only makes a decision when a process completes or a new process is added, and when a new process is added the algorithm only needs to compare the currently executing process with the new process, ignoring all other processes currently waiting to execute.

Characteristics

- Low average waiting time than SJF
- Useful in timesharing

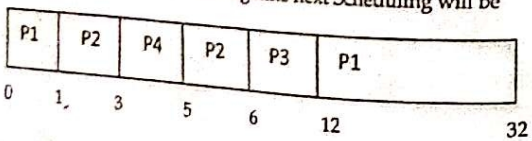
Demerits

- Very high overhead than SJF
- Requires additional computation.
- Favors short jobs, longs jobs can be victims of starvation.

Example: Consider the processes P1, P2, P3, P4 given in the below table, arrives for execution in the same order, with Arrival Time 0, 1, 2, and 3 respectively and given Burst Time, let's find the average waiting time and average turnaround time using the **Shortest remaining time next** scheduling algorithm.

Process	Burst Time	Arrival Time
P1	21	0
P2	3	1
P3	6	2
P4	2	3

The Gantt chart for Preemptive Shortest remaining time next Scheduling will be



As it is seen in the GANTT chart above, as P1 arrives first, hence it's execution starts immediately, but just after 1 ms, process P2 arrives with a burst time of 3 ms which is less than the burst time of P1, hence the process P1(1 ms done, 20 ms left) is preempted and process P2 is executed.

As P2 is getting executed, after 1 ms, P3 arrives, but it has a burst time greater than that of P2, hence execution of P2 continues. But after another millisecond, P4 arrives with a burst time of 2 ms, as a result P2 (2 ms done, 1 ms left) is preempted and P4 is executed.

After the completion of P4, process P2 is picked up and finishes, then P3 will get executed and at last P1.

Process	Burst Time	Arrival Time	Turnaround time = Completion Time – Arrival Time	Waiting Time = Turn Around Time – Burst Time
P1	21	0	32- 0=32 ms	32-21=11
P2	3	1	6-1=5 ms	5-3=2
P3	6	2	12-2=10 ms	10-6=4
P4	2	3	5-3=2 ms	2-2=0

Total waiting time: (11 + 2 + 4 + 0) = 17 ms

Average waiting time: (17/4) = 4.25 ms

Total turnaround time: (32 + 5 + 10 + 2) = 49 ms

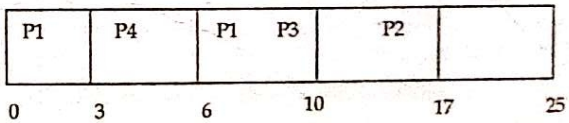
Average turnaround time: (49/4) = 12.25 ms

Example 2: Find average waiting time and average Turnaround time of following four processes with their arrival time and burst time as below,

Process	Burst Time	Arrival Time
P1	6 ms	1 ms
P2	8 ms	1 ms
P3	7 ms	2 ms
P4	3 ms	3 ms

Solution:

Gantt chart



In the above example, at time 1ms, there are two processes i.e. P1 and P2. Process P1 is having burst time as 6ms and the process P2 is having 8ms. So, P1 will be executed first. Since it is a preemptive approach, so we have to check at every time quantum. At 2ms, we have three processes i.e. P1 (5ms remaining), P2 (8ms), and P3 (7ms). Out of these three, P1 is having the least burst time, so it will continue its execution. After 3ms, we have four processes i.e. P1 (4ms remaining), P2 (8 ms), P3 (7ms), and P4 (3ms). Out of these four, P4 is having the least burst time, so it will be executed. The process P4 keeps on executing for the next three ms because it is having the shortest burst time. After

6ms, we have 3 processes i.e. P1 (4ms remaining), P2 (8ms), and P3 (7ms). So, P1 will be selected executed. This process of time comparison will continue until we have all the processes executed waiting and turnaround time of the processes will be:

Process	Waiting Time	Turnaround Time
P1	3 ms	9 ms
P2	16 ms	24 ms
P3	8 ms	15 ms
P4	0 ms	3 ms

Total waiting time: (3 + 16 + 8 + 0) = 27 ms

Average waiting time: (27/4) = 6.75 ms

Total turnaround time: (9 + 24 + 15 + 3) = 51 ms

Average turnaround time: (51/4) = 12.75 ms

Interactive System Scheduling

a. **Round Robin scheduling**

In this algorithm the process is allocated the CPU for the specific time period called time sli or quantum, which is normally of 10 to 100 milliseconds. If the process completes its executio within this time slice, then it is removed from the queue otherwise it has to wait for another time slice. Preempted process is placed at the back of the ready list.

In this approach of CPU scheduling, we have a fixed time quantum and the CPU will b allocated to a process for that amount of time only at a time. For example, if we are having three process P1, P2, and P3, and our time quantum is 2ms, then P1 will be given 2ms for it execution, then P2 will be given 2ms, then P3 will be given 2ms. After one cycle, again P1 wi be given 2 ms, and then P2 will be given 2ms and so on until the processes complete i execution.

Advantages

- Fair allocation of CPU across the process.
- Used in timesharing system.
- Low average waiting time when process lengths vary widely.
- Poor average waiting time when process lengths are identical.

Disadvantages

- We have to perform a lot of context switching here, which will keep the CPU idle

Quantum size: If the quantum is very large, each process is given as much time as needs fo completion; RR degenerate to FCFS policy. If quantum is very small, system busy at just switchin from one process to another process, the overhead of context-switching causes the system efficiency degrading.

Example: Consider the processes P1, P2, P3 given in the below table, arrives for execution in th same order, with Arrival Time 0, and given Burst Time, let's find the average waiting time using ro Average turnaround time by using the **Round Robin scheduling** algorithm.

Process	Burst Time
P1	10 ms
P2	5 ms
P3	8 ms

The Gantt chart for following processes based on Round robin scheduling with quantum size=2 will be

P1	P2	P3	P1	P2	P3	P1	P2	P3	P1	P3	P1
0	2	4	6	8	10	12	13	15	17	19	21	23

In the above example, every process will be given 2ms in one turn because we have taken the time quantum to be 2ms. So process P1 will be executed for 2ms, then process P2 will be executed for 2ms, then P3 will be executed for 2 ms. Again process P1 will be executed for 2ms, then P2, and so on. The waiting time and turnaround time of the processes will be:

Process	Turnaround Time = Completion Time – Arrival Time	Waiting Time = Turn Around Time – Burst Time
P1	23	13
P2	15	10
P3	21	13

Total waiting time: (13 + 10 + 13) = 36ms

Average waiting time: (36/3) = 12ms

Total turnaround time: (23 + 15 + 21) = 59ms

Average turnaround time: (59/3) = 19.66ms

b. **Priority scheduling**

In this scheduling algorithm the priority is assigned to all the processes and the process with highest priority executed first. Priority assignment of processes is done on the basis of internal factor such as CPU and memory requirements or external factor such as user's choice. It is just used to identify which process is having a higher priority and which process is having a lower priority. For example, we can denote 0 as the highest priority process and 100 as the lowest priority process. Also, the reverse can be true i.e. we can denote 100 as the highest priority and 0 as the lowest priority.

Advantages of priority scheduling (non-preemptive)

- Higher priority processes like system processes are executed first.

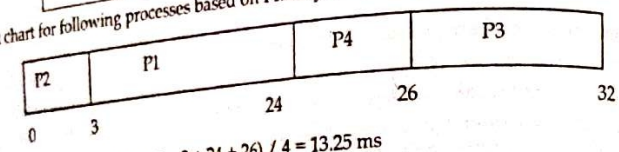
Disadvantages of priority scheduling (non-preemptive)

- It can lead to starvation if only higher priority process comes into the ready state. Low priority processes may never execute.
- If the priorities of more two processes are the same, then we have to use some other scheduling algorithm.

Example: Consider the processes P1, P2, P3, P4 given in the below table, arrives for execution in the same order, with Arrival Time 0, and given Burst Time, let's find the average waiting time using the Priority scheduling algorithm.

Process	Burst Time	Priority
P1	21	2
P2	3	1
P3	6	4
P4	2	3

The Gantt chart for following processes based on Priority scheduling will be



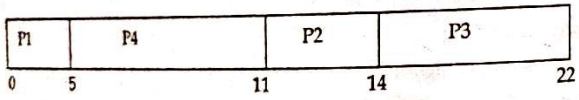
The average waiting time will be (0 + 3 + 24 + 26) / 4 = 13.25 ms

Example 2: Find average waiting time and average Turnaround time of following four process with their arrival time, burst time and priority as below,

Process	Arrival time	Burst Time	Priority
P1	0 ms	5 ms	1
P2	1 ms	3 ms	2
P3	2 ms	8 ms	1
P4	3 ms	6 ms	3

Note: in this example we are taking higher priority number as higher priority.

Gantt chart



In the above example, at 0 ms, we have only one process P1. So P1 will execute for 5ms because we are using non-preemption technique here. After 5ms, there are three processes in the ready state i.e. process P2, process P3, and process P4. Out to these three processes, the process P4 is having the highest priority so it will be executed for 6ms and after that, process P2 will be executed for 3ms followed by the process P1. The waiting and turnaround time of processes will be:

Process	Arrival time	Burst Time	Priority	Turnaround Time = Completion Time – Arrival Time	Waiting Time = Turn Around Time - Burst Time
P1	0 ms	5 ms	1	5-0=5 ms	5-5=0 ms
P2	1 ms	3 ms	2	14-1=13 ms	13-3=10 ms
P3	2 ms	8 ms	1	22-2=20 ms	20-8=12 ms
P4	3 ms	6 ms	3	11-3=8 ms	8-6=2 ms

Total waiting time: (0 + 10 + 12 + 2) = 24 ms

Average waiting time: (24/4) = 6 ms

Total turnaround time: (5 + 13 + 20 + 8) = 46 ms

Average turnaround time: (46/4) = 11.5 ms

c. **Multiple queues (Multilevel Queue Scheduling)**

It may happen that processes in the ready queue can be divided into different classes where each class has its own scheduling needs. For example, a common division is a foreground (interactive) process and background (batch) processes. These two classes have different scheduling needs. For this kind of situation Multilevel Queue Scheduling is used. Now, let us see how it works.

Ready Queue is divided into separate queues for each class of processes. For example, let us take three different types of process System processes, Interactive processes and Batch Processes. All three processes have their own queue. Now, look at the below figure.

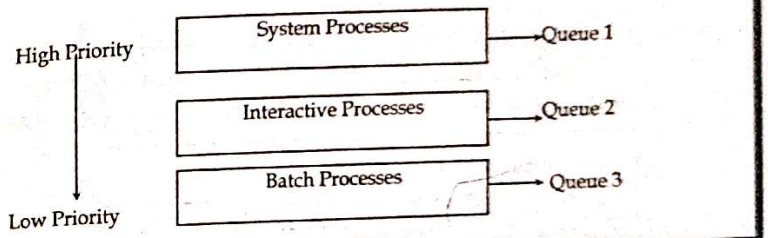


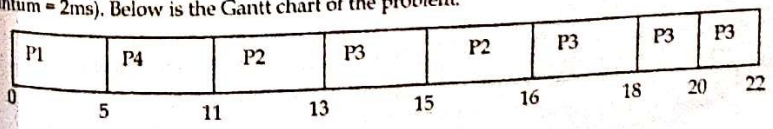
Fig 3.33: Multiple queue

All three different types of processes have their own queue. Each queue has its own Scheduling algorithm. For example, queue 1 and queue 2 uses Round Robin while queue 3 can use FCFS to schedule their processes.

Example: Consider below table of four processes under multilevel queue scheduling. Queue number denotes the queue of the process.

Process	Arrival Time	CPU Burst Time	Queue Number
P1	0	5	1
P2	0	3	2
P3	0	8	2
P4	0	6	1

In the above example, we have two queues i.e. queue1 and queue2. Queue1 is having higher priority and queue1 is using the FCFS approach and queue2 is using the round-robin approach (time quantum = 2ms). Below is the Gantt chart of the problem:



Since the priority of queue1 is higher, so queue1 will be executed first. In the queue1, we have two processes i.e. P1 and P4 and we are using FCFS. So, P1 will be executed followed by P4. Now, the of the queue1 is finished. After this, the execution of the processes of queue2 will be started by using the round-robin approach.

d. Multilevel Feedback Queue Scheduling

It allows the process to move in between queues. The idea is to separate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it is moved to a lower-priority queue. Similarly, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.

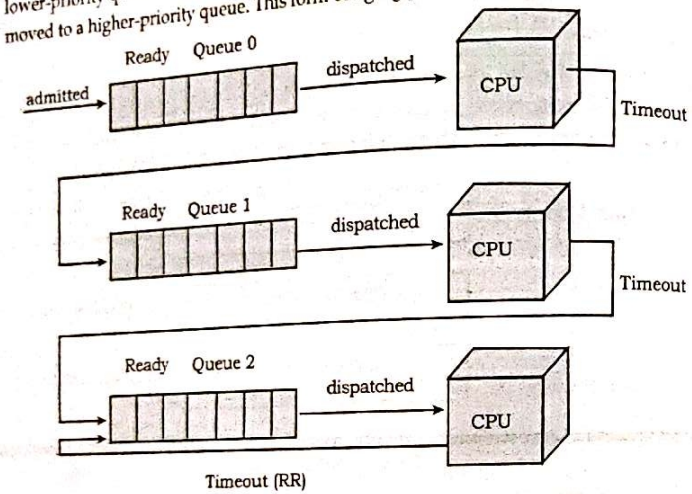
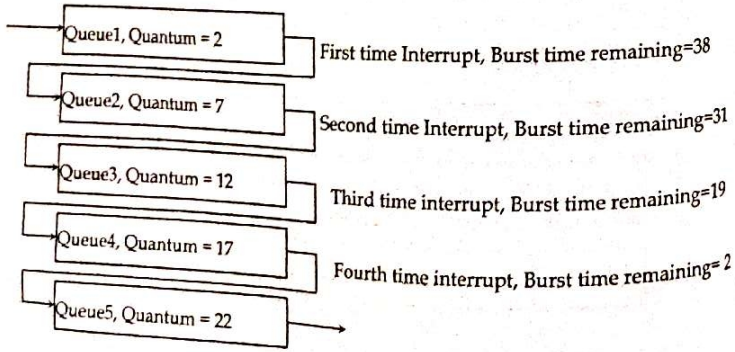


Fig 3.34: Multilevel feedback queue

Example: Consider a system which has CPU Bound process which requires burst time of 40 time units. Multilevel feedback queue scheduling is used. The time quantum is 2 units and it will be incremented by 5 units in each level. How many times the process will be interrupted and in which queue process will complete the execution?

Solution:



Hence from above figure process will be interrupted in 4 times and in 5th queue the process will complete the execution.

Practice Question 1: Consider the following set of processes, with the length of the CPU burst time given in milliseconds.

Process	Burst time	Priority
P1	10	3
P2	1	1
P3	2	3
P4	1	4
P5	5	2

The processes are assumed to have arrived in the order P1, P2, P3, P4, P5, all at time 0.

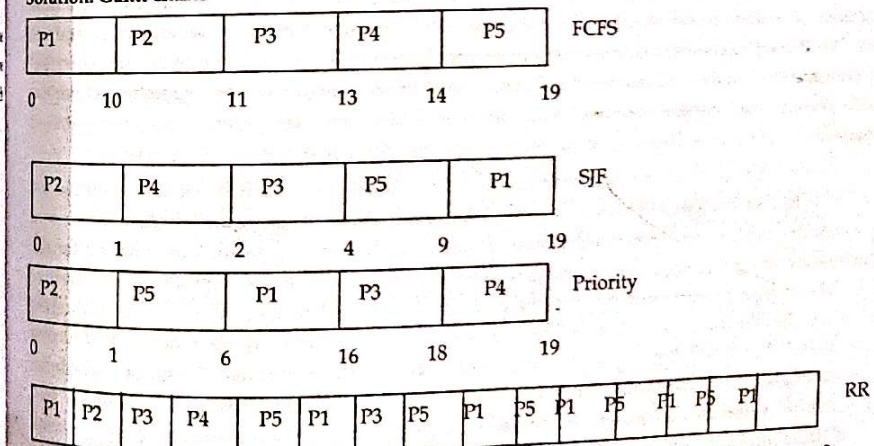
a. Draw four Gantt charts illustrating the execution of these processes using FCFS, SJF, a non- preemptive priority (a smaller priority number implies a higher priority) and RR scheduling with quantum=1.

b. What is the turnaround time of each process for each of the scheduling algorithms in part a?

c. What is the waiting time of each process for each of the scheduling algorithms in part a?

d. Which of the scheduling in part 'a' results in the minimal average time (over all processes)?

Solution: Gantt charts



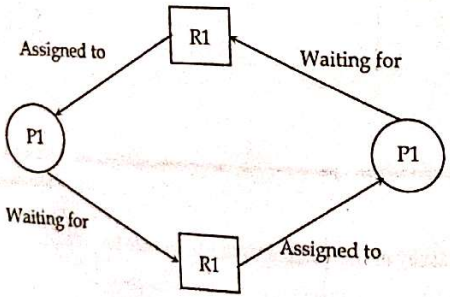
SYSTEM MODEL

A system model or structure consists of a fixed number of resources to be circulated among some opposing processes. The resources are then partitioned into numerous types, each consisting of some specific quantity of identical instances. Memory space, CPU cycles, directories and files, I/O devices like keyboards, printers and CD-DVD drives are prime examples of resource types. When a system has 2 CPUs, then the resource type CPU got two instances. Under the standard mode of operation, any process may use a resource in only the below-mentioned sequence:

- **Request**: When the request can't be approved immediately (where the case may be when another process is utilizing the resource), then the requesting job must remain waited until it can obtain the resource.
- **Use**: The process can run on the resource (like when the resource is a printer, its job/process is to print on the printer).
- **Release**: The process releases the resource (like, terminating or exiting any specific process)

INTRODUCTION TO DEADLOCK

Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process. Consider an example when two trains are coming toward each other on same track and there is only one track, none of the trains can move once they are in front of each other. Similar situation occurs in operating systems when there are two or more processes hold some resources and wait for resources held by other. For example, in the below diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1.



Now neither program can proceed until the other program releases a resource. The operating system cannot know what action to take. At this point the only alternative is to abort (stop) one of the programs.

SYSTEM RESOURCES: PREEMPTABLE AND NONPREEMPTABLE RESOURCES

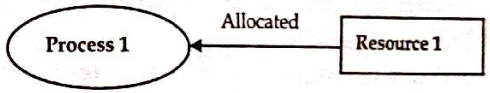
Resources come in two flavors: preemptable and nonpreemptable. A preemptable resource is one that can be taken away from the process with no ill effects. Memory is an example of a preemptable resource. On the other hand, a nonpreemptable resource is one that cannot be taken away from process (without causing ill effect). For example, CD resources are not preemptable at an arbitrary moment. Reallocating resources can resolve deadlocks that involve preemptable resources. Deadlocks that involve nonpreemptable resources are difficult to deal with.

DEADLOCK CHARACTERIZATION (CONDITIONS FOR RESOURCE DEADLOCKS)

In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting. Various features that characterize deadlock are listed below:

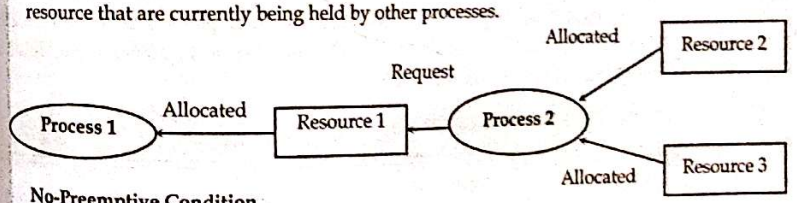
a. **Mutual Exclusion Condition**

The resources involved are non-shareable, At least one resource must be held in a non-shareable mode, i.e. only one process at a time claims exclusive control of the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.



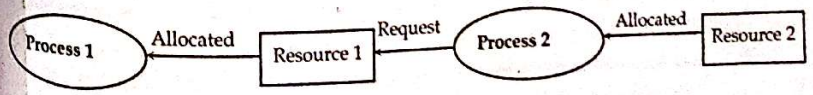
b. **Hold and Wait Condition**

Requesting process hold already, resources while waiting for requested resources, there must exist a process that is holding a resource already allocated to it while waiting for additional resource that are currently being held by other processes.



c. **No-Preemptive Condition**

Resources already allocated to a process cannot be preempted, Resources cannot be removed from the processes are used to completion or released voluntarily by the process holding it.



d. **Circular Wait Condition**

All the processes must wait for the resource in a cyclic manner where the last process waits for the resource held by the first process.

A set {P0 P1, P2... Pn} of waiting processes must exist such that P1 is waiting for a resource that is held by P1, P1 is waiting for a resource that is held by P2, ..., Pn-1 is waiting for a resource that is held by Pn and Pn is waiting for a resource that is held by P0. The processes in the system form a circular list or chain where each process in the list is waiting for a resource held by the next process in the list.

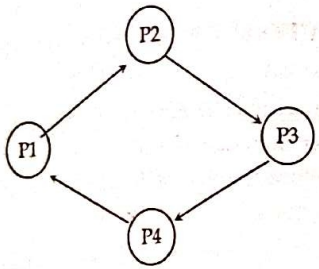


Fig: 4.1: Circular waits

DEADLOCK MODELING

The OSTRICH Algorithm

The ostrich algorithm is a strategy of ignoring potential problems on the basis that they may be exceedingly rare. It is named for the ostrich effect which is defined as "to stick one's head in the sand and pretend there is no problem". It is used when it is more cost-effective to allow the problem to occur than to attempt its prevention. This approach may be used in dealing with deadlocks in concurrent programming if they are believed to be very rare and the cost of detection or prevention is high. For example, if each PC deadlocks once per 10 years, the one reboot may be less painful than the restrictions needed to prevent it. The ostrich algorithm pretends there is no problem and is reasonable to use if deadlocks occur very rarely and the cost of their prevention would be high. The UNIX and Windows operating systems take this approach.

METHOD OF HANDLING DEADLOCKS

There are mainly four methods for handling deadlock.

a. Deadlock Prevention
b. Deadlock Avoidance
   - Banker's Algorithm
c. Deadlock Detection
   - Resource Allocation Graph
d. Recovery from Deadlock

a. **Deadlock Prevention**

It means that we design such a system where there is no chance of having a deadlock. We can prevent deadlock by eliminating any of the above four condition.

Elimination of Mutual Exclusion Condition

The mutual exclusion condition must hold for non-sharable resources. That is, several processes cannot simultaneously share a single resource. This condition is difficult to eliminate because some resources, such as the tap drive and printer, are inherently non-shareable. Note that shareable resources like read-only-file do not require mutually exclusive access and thus cannot be involved in deadlock.

Elimination of Hold and Wait Condition

There are two possibilities for elimination of the second condition. The first alternative is that a process request be granted all of the resources it needs at once, prior to execution. The second alternative is to disallow a process from requesting resources whenever it has previously allocated resources. This strategy requires that all of the resources a process will need must be requested at once. The system must grant resources on "all or none" basis. If the complete set of resources needed by a process is not currently available, then the process must wait until the complete set is available. While the process waits, however, it may not hold any resources. Thus the "wait for" condition is denied and deadlocks simply cannot occur. This strategy can lead to serious waste of resources. For example, a program requiring ten tap drives must request and receive all ten derives before it begins executing. If the program needs only one tap drive to begin execution and then does not need the remaining tap drives for several hours. Then substantial computer resources (9 tape drives) will sit idle for several hours. This strategy can cause indefinite postponement (starvation). Since not all the required resources may become available at once.

Elimination of No-preemption Condition

The non-preemption condition can be alleviated by forcing a process waiting for a resource that cannot immediately be allocated to relinquish all of its currently held resources, so that other processes may use them to finish. Suppose a system does allow processes to hold resources while requesting additional resources. Consider what happens when a request cannot be satisfied. A process holds resources a second process may need in order to proceed while second process may hold the resources needed by the first process. This is a deadlock. This strategy requires that when a process that is holding some resources is denied a request for additional resources. The process must release its held resources and, if necessary, request them again together with additional resources. Implementation of this strategy denies the "no-preemptive" condition effectively high Cost when a process release resources the process may lose all its work to that point. One serious consequence of this strategy is the possibility of indefinite postponement (starvation). A process might be held off indefinitely as it repeatedly requests and releases the same resources.

Elimination of Circular Wait Condition

The last condition, the circular wait, can be denied by imposing a total ordering on all of the resource types and then forcing, all processes to request the resources in order (increasing or decreasing). This strategy imposes a total ordering of all resources types, and to require that each process requests

resources in a numerical order (increasing or decreasing) of enumeration. With this rule, the resource allocation graph can never have a cycle. For example, provide a global numbering of all the resources, as shown

1 ≡ Card reader

2 ≡ Printer

3 ≡ Plotter

4 ≡ Tape drive

5 ≡ Card punch

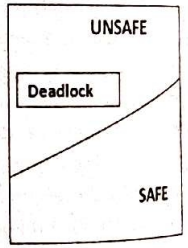
Now the rule is this: processes can request resources whenever they want to, but all requests must be made in numerical order. A process may request first printer and then a tape drive (order: 2, 4), but it may not request first a plotter and then a printer (order: 3, 2). The problem with this strategy is that it may be impossible to find an ordering that satisfies everyone.

b. Deadlock Avoidance

This approach to the deadlock problem anticipates deadlock before it actually occurs. This approach employs an algorithm to access the possibility that deadlock could occur and acting accordingly. This method differs from deadlock prevention, which guarantees that deadlock cannot occur by denying one of the necessary conditions of deadlock. If the necessary conditions for a deadlock are in place, it is still possible to avoid deadlock by being careful when resources are allocated. So named because the process is analogous to that used by a banker in deciding if a loan can be safely made.

A deadlock avoidance algorithm dynamically examines the resources allocation state to ensure that a circular wait condition case never exists. Where, the resources allocation state is defined by available and allocated resources and the maximum demand of the process. There are 3 states of the system:

- Safe state
- Unsafe state and
- Deadlock state



Safe and unsafe state

When a system can allocate the resources to the process in such a way so that they still avoid deadlock then the state is called safe state. When there is a safe sequence exit then we can say that the system is in the safe state.

A sequence is in the safe state only if there exists a safe sequence. A sequence of process P1, P2, Pn is a safe sequence for the current allocation state if for each Pi the resources request that Pi can still make can be satisfied by currently available resources pulls the resources held by all Pj with j<i.

If a safe sequence does not exist, then the system is in an unsafe state, which may lead to deadlock. All safe states are deadlock free, but not all unsafe states lead to deadlocks.

Example: Let's take four customers with available is 2

Customers	Used	Max	
A	1	6	Available
B	1	5	Units = 2
C	2	4	
D	4	7	

The key to a state being safe is that there is at least one way for all users to finish. In other analogy, the state of above table is safe because with 2 units left, the banker can delay any request except C's, thus letting C finish and release all four resources. With four units in hand, the banker can let either D or B have the necessary units and so on. Thus safe sequence is <C, D, B, A>

Consider what would happen if a request from B for one more unit were granted in above table. We would have following situation:

Customers	Used	Max	
A	1	6	Available
B	2	5	Units = 1
C	2	4	
D	4	7	

If all the customers namely *A, B, C,* and *D* asked for their maximum loans, then banker could not satisfy any of them and we would have a deadlock. Thus this is an unsafe state.

Methods for deadlock avoidance

There is mainly following Bankers algorithm is used for deadlock avoidance:

Banker's Algorithm

The Banker algorithm, sometimes referred to as the detection algorithm, is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation of predetermined maximum possible amounts of all resources, and then makes an "s-state" check to test for possible deadlock conditions for all other pending activities, before deciding whether allocation should be allowed to continue. Following **data structures** are used to implement the Banker's Algorithm:

Let '**n**' be the number of processes in the system and '**m**' be the number of resources types.

Available

- It is a one dimensional array of size 'm' indicating the number of available resources of each type.
- Available[j] = k means there are 'k' instances of resource type R_j

Max

- It is a 2-dimensional array of size 'n*m' that defines the maximum demand of each process in a system.
- Max[i, j] = k means process P_i may request at most 'k' instances of resource type R_j

Allocation

- It is a 2-dimensional array of size 'n*m' that defines the number of resources of ty type currently allocated to each process.
- Allocation[i, j] = k means process Pi is currently allocated 'k' instances of resou type Rj

Need

- It is a 2-dimensional array of size 'n*m' that indicates the remaining resource need each process.
- Need[i, j] = k means process Pi currently need 'k' instances of resource type Rj for execution.
- Need[i, j] = Max[i, j] – Allocation[i, j]

Banker's resource request Algorithm

Step 1: If request ≤ need then go to step 2

Else error

Step 2: if request ≤available then go to step 3

Else wait

Step 3: Available = available – Request

Allocation = Allocation + Request

Need = Need – Request

Step 4: Check new state is safe or not?

Bankers Safety Algorithm

Step 1: If need ≤ Available then

Execute Process

Calculate new available as,

Available= Available + Allocation

Step 2: Otherwise

- Do not execute and go forward

Numerical problem 1: Assume that there are 5 processes, P0 through P4, and 4 types of resources. T0 we have the following system state:

	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P0	0	1	1	0	0	2	1	0	1	5	2	0
P1	1	2	3	1	1	6	5	2				
P2	1	3	6	5	2	3	6	6				
P3	0	6	3	2	0	6	5	2				
P4	0	0	1	4	0	6	5	6				

Let's take a look at Need4 (0, 6, 4, 2). This is less than work, so we can update work and finish.

Work vector	Finish matrix	
1	P0	TRUE
10	P1	FALSE
6	P2	FALSE
6	P3	TRUE
	P4	TRUE

We can now go back to P1. Need1 (0, 2, 1, 1) is less than work, so work and finish can be updated

Work vector	Finish matrix	
1	P0	TRUE
14	P1	FALSE
10	P2	FALSE
7	P3	TRUE
	P4	TRUE

Finally, Need2 (1, 0, 0, 1) is less than work, so we can also accommodate this. Thus, the system is safe state when the processes are run in the following order: {P0, P3, P4, P1, P2}. We therefore grant the resource request.

Numerical problem 2: Assume that there are three resources, A, B, and C. There are 4 processes to P3. At T0 we have the following snapshot of the system:

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P0	1	0	1	2	1	1	2	1	1
P1	2	1	2	5	4	4			
P2	3	0	0	3	1	1			
P3	1	0	1	1	1	1			

1. Create the need matrix (max-allocation)

	Need matrix(max-allocation)		
	A	B	C
P0	1	1	0
P1	3	3	2
P2	0	1	1
P3	0	1	0

Note: If each resource has only one instance, the cycle always causes the deadlock, if each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In the graph represented on right P4 may release its instance of resource type R2 and P3 is allocated breaking the cycle.

d. **Deadlock Recovery (Recovery from Deadlock)**

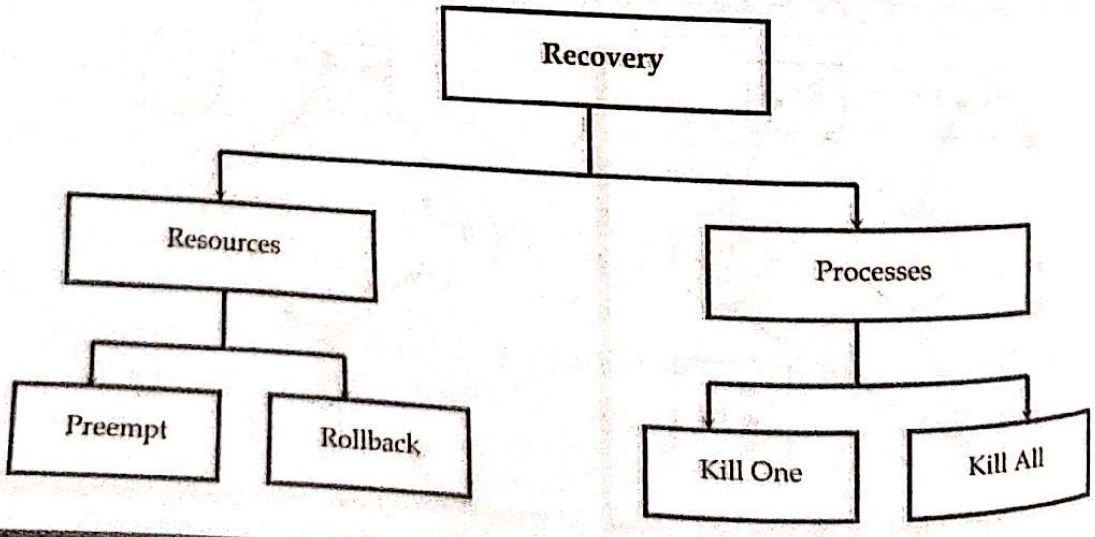
Traditional operating system such as Windows doesn't deal with deadlock recovery as it is time and space consuming process. Real time operating systems use Deadlock recovery. Some of the common recovery methods are

For Resource

- **Preempt the resource:** We can snatch one of the resources from the owner of the resource (process) and give it to the other process with the expectation that it will complete the execution and will release this resource sooner. Well, choosing a resource which will be snatched is going to be a bit difficult.
- **Rollback to a safe state:** System passes through various states to get into the deadlock state. The operating system can roll back the system to the previous safe state. For this purpose, OS needs to implement check pointing at every state. The moment, we get into deadlock, we will rollback all the allocations to get into the previous safe state.

For Process

- **Kill a process:** Killing a process can solve our problem but the bigger concern is to decide which process to kill. Generally, operating system kills a process which has done least amount of work until now.
- **Kill all process:** This is not a suggestible approach but can be implemented if the problem becomes very serious. Killing all process will lead to inefficiency in the system because all the processes will execute again from starting.



It is important to note that an unsafe state does not imply the existence or even the eventual existence of a deadlock. What an unsafe state does imply is simply that some unfortunate sequence of events might lead to a deadlock. The Banker's algorithm is thus to consider each request as it occurs, and see if granting it leads to a safe state. If it does, the request is granted, otherwise, it postponed until later.

c. **Deadlock Detection**

Deadlock detection is the process of actually determining that a deadlock exists and identifying the processes and resources involved in the deadlock. The basic idea is to check allocation against resource availability for all possible allocation sequences to determine if the system is in deadlocked state a. Of course, the deadlock detection algorithm is only half of this strategy. Once a deadlock is detected, there needs to be a way to recover several alternatives exists:

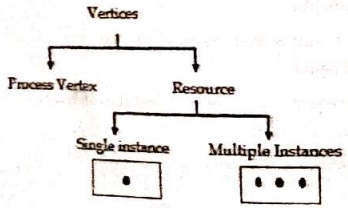
- Temporarily prevent resources from deadlocked processes.
- Back off a process to some check point allowing preemption of a needed resource and restarting the process at the checkpoint later.
- Successively kill processes until the system is deadlock free.

These methods are expensive in the sense that each iteration calls the detection algorithm until system proves to be deadlock free. The complexity of algorithm is $O(N^2)$ where N is the number of proceeds. Another potential problem is starvation; same process killed repeatedly.

Resource allocation graph (RAG)

Resource allocation graph is explained as what is the state of the system in terms of processes and resources. Like how many resources are available, how many are allocated and what is the request of each process. Everything can be represented in terms of the diagram. One of the advantages of having a diagram is, sometimes it is possible to see a deadlock directly by using RAG, but then you might not be able to know that by looking at the table. But the tables are better if the system contains lots of process and resource and Graph is better if the system contains less number of process and resource. We know that any graph contains vertices and edges. So RAG also contains vertices and edges. In RAG vertices are two types:

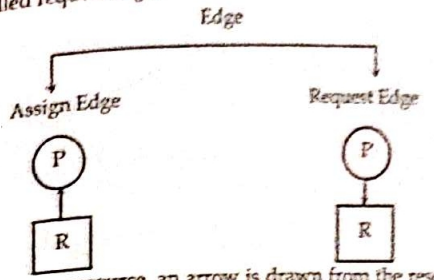
1. **Process vertex** – Every process will be represented as a process vertex. Generally, the process will be represented with a circle.
2. **Resource vertex** – Every resource will be represented as a resource vertex. It is also two type
   - **Single instance type resource** – It represents as a box, inside the box, there will be one dot. So the number of dots indicates how many instances are present of each resource type.
   - **Multi-resource instance type resource** – It also represents as a box, inside the box, there will be many dots present.



There are two types of edges in RAG

Assign Edge: *If you already assign a resource to a process then it is called Assign edge.*

Request Edge: *It means in future the process might want some resource to complete the execution that is called request edge.*



So, if a process is using a resource, an arrow is drawn from the resource node to the process node. If a process is requesting a resource, an arrow is drawn from the process node to the resource node.

Example 1 (Single instances RAG)

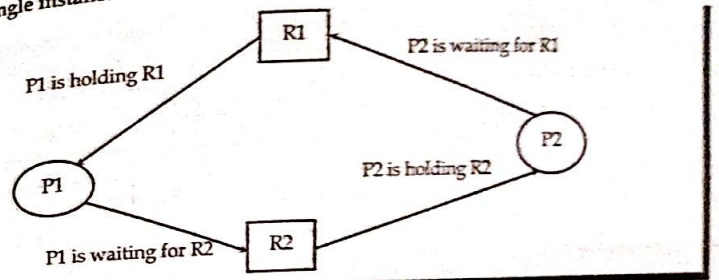


Fig 4.2: Single Instance Resource Type with Deadlock

If there is a cycle in the Resource Allocation Graph and each resource in the cycle provides only one instance, then the processes will be in deadlock. For example, if process P1 holds resource R1, process P2 holds resource R2 and process P1 is waiting for R2 and process P2 is waiting for R1, then process P1 and process P2 will be in deadlock.

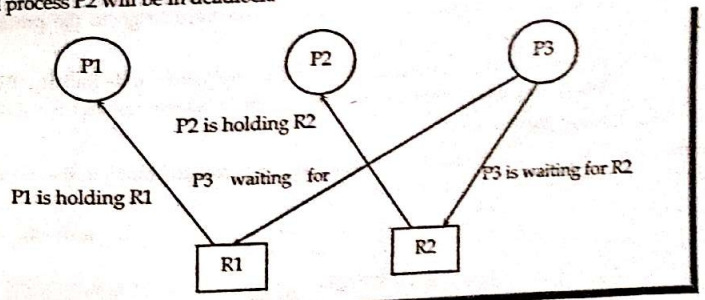


Fig 4.3: Single instance resource type without deadlock

Here is another example that shows Processes P1 and P2 acquiring resources R1 and R2 while process P3 is waiting to acquire both resources. In this example, there is no deadlock because there is no circular dependency. So cycle in single-instance resource type is the sufficient condition for deadlock.

Example 2 (Multi-instances RAG)

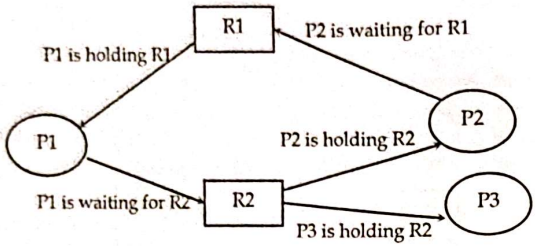


Fig 4.4: Multi Instances without Deadlock

From the above example, it is not possible to say the RAG is in a safe state or in an unsafe state. So to see the state of this RAG, let's construct the allocation matrix and request matrix.

Process	Allocation Resources		Request Resources	
	R1	R2	R1	R2
P1	1	0	0	1
P2	0	1	1	0
P3	0	1	0	0

- The total number of processes are three; P1, P2 & P3 and the total number of resources are two; R1 & R2.

Allocation matrix

- For constructing the allocation matrix, just go to the resources and see to which process it is allocated.
- R1 is allocated to P1, therefore write 1 in allocation matrix and similarly, R2 is allocated to P2 as well as P3 and for the remaining element just write 0.

Request matrix

- In order to find out the request matrix, you have to go to the process and see the outgoing edges.
- P1 is requesting resource R2, so write 1 in the matrix and similarly, P2 requesting R1 and for the remaining element write 0.So now available resource is = (0, 0).

Checking deadlock (safe or not)

Available = [0 0] (As P3 does not require any extra resources to complete the execution and after completion P3 [0 1] P3 release its own resources)

New Available = [0 0] (As using new available resource we can satisfy the requirement of process P1 and P1 also P1 [1 0] release its previous resource)

New Available = [1 1] (Now easily we can satisfy the requirement of Process P2)

P2 [0 1]

New Available = [1 2]

So there is no deadlock in this RAG. Even though there is a cycle, still there is no deadlock. Therefore, in multi-instance resource cycle is not sufficient condition for deadlock.

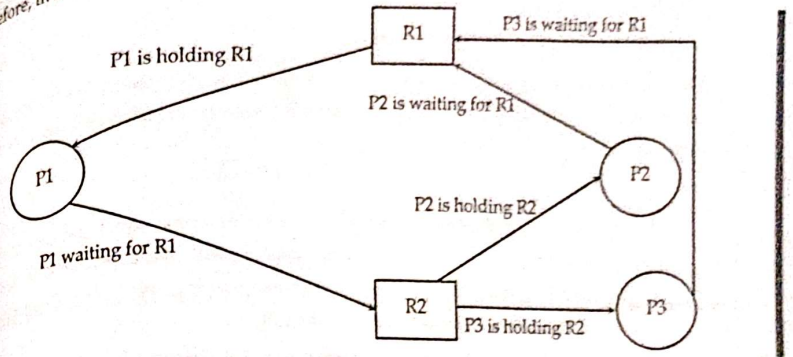


Fig 4.5: Multi Instances with deadlock

Above example is the same as the previous example except that, the process P3 requesting for resource R1.So the table becomes as shown in below.

Process	Allocation Resources		Request Resources	
	R1	R2	R1	R2
P1	1	0	0	1
P2	0	1	1	0
P3	0	1	1	0

So, the Available resource is = (0, 0), but requirement are (0, 1), (1, 0) and (1, 0).So you can't fulfill any one requirement. Therefore, it is in deadlock. Therefore, every cycle in a multi-instance resource type graph is not a deadlock, if there has to be a deadlock, there has to be a cycle. So, in case of RAG with multi-instance resource type, the cycle is a necessary condition for deadlock, but not sufficient.

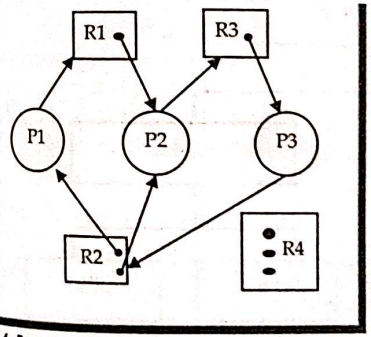


Fig 4.6: Resource allocation Graph with a deadlock

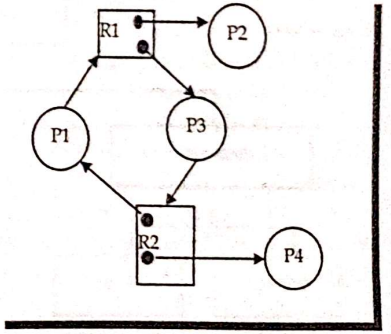


Fig 4.7: Resource allocation Graph with a cycle but no deadlock